DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MANAGEMENT ASPECTS
OF
SOFTWARE MAINTENANCE

by

Brian J. Henderson
and
Brenda J. Sullivan

September 1984

Thesis Advisor: Carl R. Jones

Approved for public release; distribution unlimited

T222095

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Management Aspects of Software Maintenance		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis September 1984 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Brian J. Henderson and Brenda J. Sullivan		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE September 1984 13. NUMBER OF PAGES 115
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

software maintenance, software management, software cost estimating, software tools

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The Federal government depends upon software systems to fulfill its missions. These software systems must be maintained and improved to continue to meet the growing demands placed on them. The process of software maintenance and improvement may be called "software evolution". The software manager must be educated in the complex nature of software maintenance to be able to properly evaluate and manage the software maintenance effort. In this thesis, the authors explore software maintenance (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Abstract (Continued)

from a management perspective, highlighting topics of critical importance. These topics include forecasting software maintenance, estimating the resources required to perform software maintenance, managing maintenance personnel and effectively utilizing software tools. The synthesis of these topics forms a managerial paradigm for understanding the evolutionary nature of software maintenance.

Approved for public release; distribution unlimited.

Management Aspects of Software Maintenance

by

Brian J. Henderson
Lieutenant, United States Navy
B.A., University of Washington, 1979

and

Brenda J. Sullivan
Lieutenant, United States Navy
B.S., The American University, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
September 1984

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943

ABSTRACT

The Federal government depends upon software systems to fulfill its missions. These software systems must be maintained and improved tc continue to meet the growing demands placed on them. The process of software maintenance and improvement may be called "software evolution". The software manager must be educated in the complex nature cf software maintenance to be able to properly evaluate and manage the software maintenance effort. In this thesis, the authors explore software maintenance from a management perspective, highlighting topics of critical impcrtance. These topics include forecasting software maintenance, estimating the resources required to perform software maintenance, managing maintenance personnel and effectively utilizing software tcols. The synthesis of these topics forms a managerial paradigm for understanding the evolutionary nature of software maintenance.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL

TABLE OF CONTENTS

I. INTRODUCTION 10
A. BACKGROUND 10
B. PROBLEM DISCUSSION 12
C. GENERAL PROCEDURE 14
D. ORGANIZATION 14

II. SOFTWARE MAINTENANCE DEFINED 16
A. THE NATURE OF SOFTWARE 16
B. SOFTWARE MAINTENANCE ACTIVITIES 19
C. A DEFINITION OF SOFTWARE MAINTENANCE 20
D. SOFTWARE MAINTENANCE AND THE SOFTWARE LIFE CYCLE 22
E. LAWS OF PROGRAM EVOLUTION AND MAINTENANCE . . 29

III. FORCASTING MAINTENANCE 32

IV. DATA REQUIRED FOR MAINTENANCE COST ESTIMATION . . 35
A. SOFTWARE CHARACTERISTICS 37
1. Development History 37
2. Maintenance History 37
3. Type of Program 38
4. Complexity 39
B. ENVIRONMENTAL CHARACTERISTICS 43
1. Personnel 43
2. Computer Attributes 46
3. Software Tools 48
4. Programming Techniques and Standards . . . 48
5. Data Base 48
C. RECOMMENDATIONS 48

V. MAINTENANCE COST ESTIMATION 50
A. OVERVIEW 50
B. TRADITIONAL METHODS 50
C. PARAMETRIC MODELS 53
D. ESTIMATING MAINTENANCE COSTS 57
1. Planning an Estimate 57
2. Evaluating a Software Maintenance Cost Model 57
E. DEPARTMENT OF DEFENSE AND SOFTWARE COST ESTIMATING 59
F. THE DEATH OF SOFTWARE 63

VI. PERSONNEL CONSIDERATIONS 68
A. INTRODUCTION 68
B. SKILLS AND EXPERIENCE NEEDED IN SOFTWARE MAINTENANCE 68
1. Military 72
2. Civil Service 73
3. Contractors 74
C. PERSONNEL ATTRIBUTES 74
D. A MAINTENANCE PROGRAMMER PERSONALITY PROFILE 76
E. ORGANIZATION 77

VII. TOOLS AND STANDARDS 80
A. INTRODUCTION 80
B. SYSTEM VIEW 80
1. Integration 81
2. Support 81
3. Standardization 81
4. Support of Standard Languages 81
5. Flexibility and Maintainability 81
C. TOOLS 81
D. TYPES OF TOOLS 82

E. ENVIRONMENTS 85
1. Programming Manager 86
2. Ada Programming Support Environment . . . 87
F. USE OF TOCLS AND STANDARDS 90

VIII. DATA EVOLUTION 94
A. DATA AS A TOOL 94
B. USE OF DATA BASE MANAGEMENT SYSTEMS 95
1. Class I Environment: Files 96
2. Class II Environment: Application Data Base 97
3. Class III Environment: Subject Data Bases 98
4. Class IV Environment: Information Systems 99
5. Class V Environment: Distributed Data Base 100
C. INDIVIDUAI DATA NEEDS 101

IX. CCNCLUSIONS/RECOMMENDATIONS 102
A. THE PROBLEM 102
B. CONCLUSIONS 103
1. Historical Data Collection 103
2. Predicting Software Maintenance: 104
3. Personnel 104
4. Tools 105
5. Summary 106

APPENDIX A: TOOLS 107
A. TOOL CATAICGS AND REFERENCES 107
B. SOFTWARE MAINTENANCE COST ESTIMATING MODELS 108

LIST CF REFERENCES 110

BIBLIOGRAPHY 114

INITIAL DISTRIBUTION LIST 115

LIST OF TABLES

I. Ccrrective vs. Enhancement Maintenance 21
II. Scftware Cost Data Elements 36
III. Module Complexity Rating vs Type of Module 42
IV. Maintenance-Critical Documentation 44
V. Software Maintenance Functions 51
VI. Software Maintenance Cost Estimating Procedure . . 58
VII. Model Parameters for Requirements Analysis Phase 60
VIII. Mcdel Parameters in Specification and Design Phase 61
IX. Model Parameters in Development and Maintenance Phase 62
X. Tool Function Taxonomy 84
XI. Software Quality Measurement Tools 85
XII. Approved and Ncnapproved Data Management Languages 93

LIST OF FIGURES

2.1 Software Maintenance Activities 20
2.2 Software Maintenance Life Cycle 25
2.3 Software Life Cycle - Putnam 28
2.4 Software Life Cycle - McClure 29
2.5 Software Life Cycle - Reality 30
6.1 Communication Styles 78
7.1 The Ada Programming Support Environment 88

I. INTRODUCTION

A. BACKGROUND

The federal government for the last twenty to thirty years has become more and more reliant on computer processing to accomplish its seemingly ever increasing and complex missions. In 1955, when the trend started, hardware was the overriding concern, consuming 85% of the total computing dollar [Ref. 1: p. 18]. Since that time, however, dramatic improvements in technology and production have substantially decreased the cost of computer hardware. Software, on the other hand, has not benefitted from technological advancements to the same degree as hardware and has continued to rise in price relative to hardware. The rise in the price of software and the decrease in the price of hardware has resulted in software rapidly becoming the more costly of the two. It is predicted that by 1985 software costs will dominate hardware costs by a ratio of nine to one [Ref. 1: p. 18]. The true impact of this trend becomes significant when one realizes that the annual cost of software (development and maintenance) in the United States in 1980 was about $40 billion, or about 2% of the Gross National Product [Ref. 1: p. 17]. It is predicted that by 1985 annual software costs will reach $200 billion [Ref. 1: p. 18].

A significant share of these costs are for software maintenance. Various studies have shown that from forty - to - seventy percent of the manpower effort in most ADP activities is dedicated to software maintenance [Ref. 1, 2, 3]. Despite its monetary significance, there is as yet no universally agreed upon definition of software maintenance.

The extensive research done on software development and on the management of the development process is only now beginning to have its counterpart in the field of software maintenance. The underlying nature and causes of software maintenance are still imperfectly understood by management at all levels, military and industry. The reasons for this lack of understanding [Ref. 4: p. 2-12] include:

1. **Executive decision makers' lack of computer related experience:** For a manager overseeing software maintenance this lack of experience is often demonstrated through impatience with system limitations and intolerance for the costs of system enhancements.
2. **Hardware orientation of software management mechanisms:** Most directives and techniques for controlling the development and maintenance of software have been adopted from hardware engineering disciplines. Thus, quality assurance, reliability and maintainability, and configuration management procedures reflect an orientation toward tangible products. Their translation for use with the environment of intangible software components has not been a completely successful one.
3. **Development vice life cycle focus:** This has significant impact on the tasks of managing and maintaining software after development. Computer programs that are developed in the most expeditious, cost-effective way to meet performance standards are not necessarily conducive to maintainability. Often the development project manager must sacrifice software design features that are conducive to program maintainability in order to meet cost, schedule or performance requirements. Thus the user is left with software that is costly to maintain.

4. Increased software system complexity: Complexity is not inherently bad for maintenance if introduced in moderation and if documentation is adequate. In today's data processing environment there is less need than ever before for complex designs and elegant code. Considering the increasing costs of software development and maintenance it makes more sense to produce straightforward program logic and code.
5. "Low-bid" contracting for acquisition of a software system: This situation affects maintenance indirectly as a result of the efforts of any cost cutting on the part of the developer. Given the degree to which DoD contracts its software development, this problem has significant impact on the military.
6. Risk, cost and reliability estimating deficiencies: Accurate estimation techniques would greatly enhance the maintenance managers effectiveness in allocating resources for program maintenance
7. Absence of Common Software Maintenance Practices: Management at all levels are placed in the awkward position of having to learn to interpret management control data from each new system.

B. PROBLEM DISCUSSION

This thesis will study software maintenance from a management perspective. Primary emphasis will be placed on examining pertinent aspects of the management of the function of software maintenance. The thesis will focus on the maintenance activity itself, rather than on the interface between the activity and the users of software. The management of that interface is termed "Configuration Management", and is well-governed with numerous policies and standards. The majority of existing software configuration management

doctrine focuses on software development, while providing little assistance to the software maintenance manager. The authors do not intend to present a "how-to" manual for software maintenance; rather, a framework will be offered upon which the manager may develop his or her understanding.

A central premise of this thesis is that software evolves. The concept of software evolution has been explored in the literature before [Ref. 5: p.217] and provides the basis for a paradigm with which a software manager may understand the nature and causes of software maintenance.

Software evolution is influenced by a number of internal and external factors. External factors define the environment to which a given software system must adapt, and internal factors define the ability of the system to make the adaptation. The goal of the software manager is to direct the evolution of the software toward a system that continues to meet organizational goals, or at least away from a system that is inefficient and expensive.

The software manager must seek to understand the factors that influence software evolution in order to achieve the goal of directing that evolution. By understanding these factors, he or she may then learn to predict their influence on software evolution. Once the influence of the internal and external factors may be predicted, the software manager may then seek to control those factors and direct the evolution of the software system.

A failure of the manager to even understand how and why software evolves will allow the software system to evolve in an uncontrolled fashion towards a morass of inflexible and unreliable "spaghetti" code. Controlling the evolution of software allows the software manager to maintain a functioning, effective software system well into the future.

Software, like any evolving entity, may reach an evolutionary dead-end. This occurs when the internal factors (code structure and design) make it impossible to respond to the evolutionary demands of external factors. Software in this stage may be said to have achieved "software senility".

The intent of this thesis is to help the software manager understand what factors influence software evolution, how to predict software evolution, and finally, some ideas on how to control the influence of internal and external factors.

C. GENERAL PROCEDURE

The procedure used was to research literature concerning software maintenance. Particular emphasis was placed on software maintenance management, cost estimation, and methodologies to conduct software maintenance. The personal experience of LT Sullivan was invaluable in placing much of the research in perspective.

D. ORGANIZATION

Chapter II develops a definition of software maintenance, and discusses the major activities conducted during maintenance. The similarities of software maintenance to software development and the characteristics of the maintenance phase of the software life cycle are also discussed. Considerations in predicting required software maintenance are explored in Chapter III, and the data required to accurately predict software maintenance is discussed in Chapter IV. In Chapter V, methods of estimating software maintenance costs are presented, and problems associated with current estimating techniques are discussed. Chapter VI explains in more detail personnel consideration in software maintenance, and Chapter VII explores the impact of software

tools and standards. The relationship of software maintenance and data is the subject of Chapter VIII. Chapter IX summarizes the authcrs' views on software maintenance, explains a paradigm with which a software manager may better understand software maintenance.

II. SOFTWARE MAINTENANCE DEFINED

A. THE NATURE OF SOFTWARE

Software may be defined as "a realization of a set of plans or specifications, encoded in computer language." [Ref. 6: p. 7]. Software is not a physical entity, it is an abstraction, a logical representation that is physically manifested in the form of program listings and documentation. Software, unlike hardware, does not wear out. Hardware is subject to deterioration in the course of normal operation and requires maintenance in order to restore it to its former operating condition. Software, on the other hand, does not change unless and until people change it. Software does not wear out of its own accord. Software maintenance does not mean restoring software to its former state, rather it involves changes away from the previous implementation. In the case of hardware, the former operating condition was the ideal and deterioration has caused degraded performance. Restoration of hardware to the former operating condition will restore optimum performance. With software, however, defects or deficiencies in the former state will have caused degraded performance, and software must be changed to a state different from the original in order to restore optimum performance. Software maintenance becomes a process in which the software is continually changed in order that its performance may be improved or maintained. Unfortunately, software maintenance is often so poorly done that the software's performance is neither maintained or improved. The nature of software maintenance is well-summarized below:

> Unfortunately, the nature of hardware and software errors differ in at least one fundamental characteristic - hardware deteriorates because of a lack of maintenance, whereas software deteriorates because of the presence of maintenance [Ref. 7: p. 11].

A landmark study of software maintenance is that done by Bennet P. Lientz and E. Burton Swanson [Ref. 2]. In it the authors specified three basic categories of software maintenance:

1. Corrective maintenance: Emergency program fixes and routine debugging.
2. Adaptive: The accommodation of changes to data inputs and fields, and to hardware and software.
3. Perfective maintenance: Enhancements for users, improvements of program documentation, and recoding for efficiency in computation. [Ref. 2: p. 68].

A 1982 study by Rome Air Development Center (RADC) grouped software maintenance into four basic categories [Ref. 6]. While very similar to the categories of [Ref. 2], the RADC study included a category of "modifying" maintenance.

4. Modifying: Requirements or specifications are changed. These changes may result from inadequate initial analysis and specifications; they may spring from new insights or better ideas about the requirements and specifications, or they may be caused by evolving applications and environments.

A General Accounting Office (GAO) study [Ref. 8: pp. 28-29] offered six categories of software maintenance:

1. Modify or enhance the software to make it do things for the end user that were not requested in the original system design.
2. Modify or enhance software to make it do things for the end users that were called for in the original design but which were not present in the first production version of the software.

3. Remove defects in which the software does something other than what the user wanted.
4. Remove defects in which the software is programmed incorrectly.
5. Optimize the software to reduce the machine cost of running it, leaving the user results unchanged.
6. Make miscellaneous modifications, such as those needed to interface with new releases of operating systems.

The various categories of software maintenance may be abstracted into two broad categories:

- Corrective: Corrective maintenance may be characterized as modifications that leave the functional specifications of the system unchanged. Such maintenance is necessary and mandatory, in the sense that the system cannot operate or existing specifications cannot be met. This would include corrective and adaptive maintenance categories of Lientz and Swanson and RADC, and categories 3 through 6 from the GAO study.

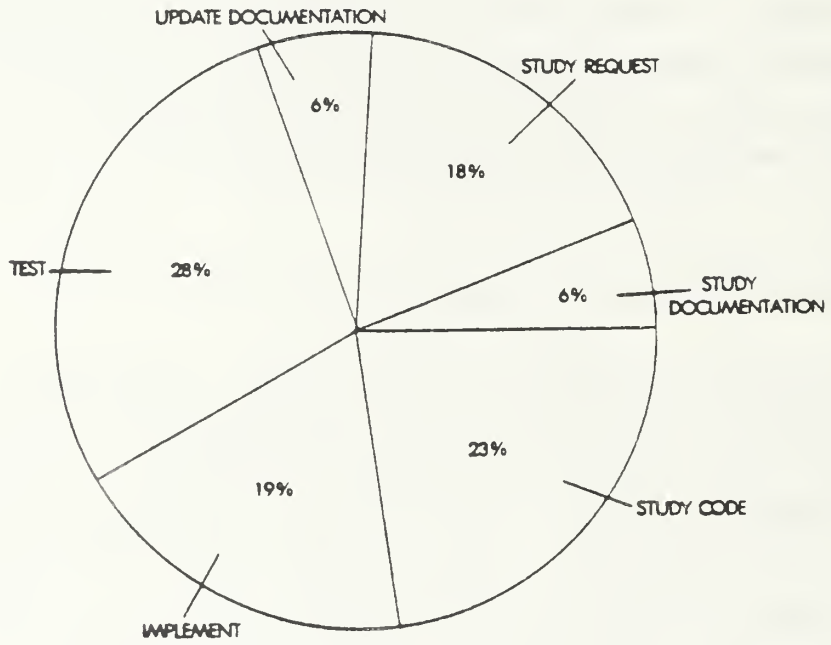
- Enhancement: Enhancement maintenance changes the original functional specifications of the system but leaves the primary functions intact. That is to say, an enhancement may add a report that was not called for in the original specifications but which is now required by a user due to changed government reporting regulations, but an enhancement does not change a payroll system to comprehensive management system integrating payroll, accounting and inventory functions. Two maintenance activities not specifically included in previous categories are maintenance due to a growth in the system or as a response to changing requirements. Growth of a system includes

expansion of the number of users serviced or files generated and accessed. Changing requirements are represented by the changed government regulations example, such as the proposed nine-digit Zip Code change. Enhancement maintenance is considered largely discretionary. This would include Perfective maintenance category of Lientz and Swanson, perfective and modifying categories of RADC, and categories 1 and 2 from the GAO study.

B. SOFTWARE MAINTENANCE ACTIVITIES

A popular misconception about software maintenance, one reinforced by the use of the term "maintenance", is that the primary activity is the correction of "bugs". The three studies discussed earlier revealed that correcting bugs is a small part of the actual maintenance effort. Figure 2.1 shows the distribution of software maintenance activities in the organizations studied in [Ref. 2], while Table I compares corrective and enhancement maintenance percentages for each of the three studies cited.

Successful software maintenance depends upon gaining a level of understanding of the software system. Software cannot be maintained unless those responsible for maintenance understand the software. Maintenance personnel spend at least half of their time trying to understand - the system code, the system documentation, and the requests from the users. Figure 2.1 shows data on the activities of maintenance personnel in performing an enhancement. Maintenance personnel spend about 47% of their time studying when making an enhancement, and about 62% when making a correction [Ref. 9: p. 2]. In a study of application program maintenance it was observed that:



[Ref. 9: p.2]

Figure 2.1 Software Maintenance Activities.

Understanding the intent and style of implementation of the original programmer was the major cause of time and difficulty in making the change [Ref. 10: p. 8].

C. A DEFINITION OF SOFTWARE MAINTENANCE

The authors' research has yielded numerous definitions of software maintenance that encompass some or all of the above named categories. The definitions differ in the manner in which they treat the abstracted categories of

TABLE I

Corrective vs. Enhancement Maintenance

	Corrective	Enhancement
Lientz & Swanson	17	64
GAO	19	51
RADC	31	61

note: Corrective maintenance figures do not include adaptive maintenance

[Ref. 9: p. 1, 6: p. 27]

"corrective maintenance" and "enhancement maintenance". A definition of software maintenance that includes both corrective and enhancement categories is termed an "inclusive" definition. A definition that includes corrective but not enhancement is an "exclusive" definition. Enhancement maintenance in this context is termed "continued development", or perhaps "production programming".

Software maintenance, for the purposes of this thesis, will be defined as:

> ....all those activities associated with a software system after the system has been initially defined, developed, deployed and accepted [Ref. 6: p. 9].

This may be summarized as the "function of keeping software in an operational mode" [Ref. 11: p. 139]. This inclusive definition is used because:

1. Both corrective and enhancement maintenance are performed by the same organization, and often by the same person. Approximately two-thirds of the systems studied in [Ref. 2] were maintained by one or two people. Both forms of software maintenance are performed concurrently in the same environment using the same tools.
2. The bulk of the effort in software maintenance is in understanding the software.
3. The term "maintenance" has been accepted as referring to both correction and enhancement, despite the poor connotation of the word.
4. The inclusive definition reflects the concept of software evolution. With the inclusive definition software may gradually evolve from the original product, rather than being continually redefined and redeveloped.
5. It is difficult to say where the separation between corrective maintenance and continued development would occur. Given that both activities are usually performed by the same person, such a distinction becomes meaningless.

D. SOFTWARE MAINTENANCE AND THE SOFTWARE LIFE CYCLE

The software life cycle is the multiphase process beginning with problem definition and continuing to software system obsolescence. The software life cycle is separated into two primary phases, the development phase and the maintenance phase. While there is some debate over the validity of this separation given the evolving, continually developing nature of software systems, it will be adhered to in this thesis because it supports the accepted inclusive definition of software maintenance. The sub-phases of the development phase are:

1. Requirements analysis: The objective of this stage is to define the requirements of a software system. Resources such as manpower and hardware and software support needed to create and support the software are considered.
2. Specification: The stage in which each function to be performed by the software is precisely defined.
3. Design: The stage in which algorithms are developed to describe how each specific software system function is to be performed.
4. Coding: The stage in which the design algorithms are translated into computer code. The translation of the design into code must be such that the resultant software neither adds nor subtracts form the design definition.
5. Testing: The objective of this stage is to demonstrate that the software conforms to specifications and performs correctly for all input data. The goal of testing is to eliminate unexpected program conditions and failures and to discover any incorrect implementation of the specification [Ref. 11: p. 32].

A software system that is designed with future maintenance in mind will more readily evolve. The three principles of maintainable software that should be embodied in the original design are:

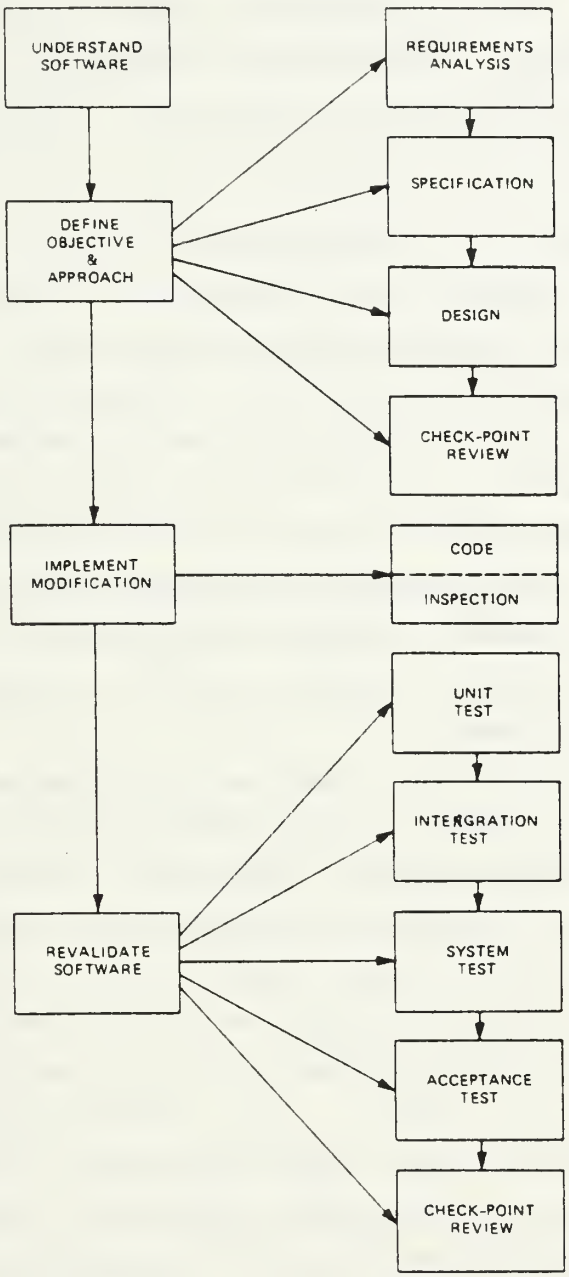
- Understandability: The ease with which software code and documentation may be read and understood.
- Testability: The ease with which the correctness of changes may be demonstrated.
- Modifiability: The ease with which software code may be modified. [Ref. 11: pp. 36-37].

The maintenance phase is very similar to the development phase with the exception of the initial stage of

understanding the software. Figure 2.2 shows the maintenance life cycle. All aspects of the modification approach must be considered in the context of the existing installed software, not just in terms of the structural, human engineering, reliability, and efficiency factors that are the major considerations when developing software. The maintenance objective is to limit the effect of a modification on other parts of the installed software and on user interfaces, to avoid excessive confusion and retraining as well as to avoid compromising system integrity and quality. Once an understanding of the software is gained, the maintenance phase, particularly in the case of enhancement maintenance, proceeds as a microcosm of the development phase.

The stages of the software maintenance phase may be defined as follows:

1. Understand the Software: During this stage the software system program listings and available documentation are studied in order to gain an understanding of the system's logic and processes. The user's complaint of error or request for modification is also studied in order to determine what action needs to be taken.
2. Define Objective and Approach: This stage includes:
   a) Requirements Analysis: The system capabilities and the resources needed to provide the modification are defined in the context of existing system capabilities and constraints.
   b) Specification: Each new function to be performed by the software modification and the impact on existing functions is precisely defined.
   c) Design: Changes to the design algorithms and procedures are defined, or, in the case of poor documentation, new algorithms are developed to describe how each new or modified function is to be performed.



[Ref. 11: p. 151]

Figure 2.2 Software Maintenance Life Cycle.

d) Check-point review: This step affords a chance to validate and verify the proposed modification. The software manager must evaluate whether the proposed modification accurately and completely addresses the problem, and whether the cost and impact of the modification justifies implementation.

3. Implement the Modification: This is the coding stage, where the modification design is correctly translated into well-structured code.

4. Revalidating the Software: During this stage it must be demonstrated that the modifications are correctly implemented, that the software system as a whole still functions correctly, and that software quality has not been harmed by the modification. The actual testing of the software modification and its impact on the system follows from the testing steps of the design process:

   a) Unit testing: Each module changed is unit tested to determine if it functions properly.

   b) Integration Test: Regression testing is performed as each module is re-integrated into the system to determine if any other parts of the system have been adversely affected by the modification.

   c) System and acceptance Test: The changed system is tested to ensure that it meets both the original design and the modification specifications.

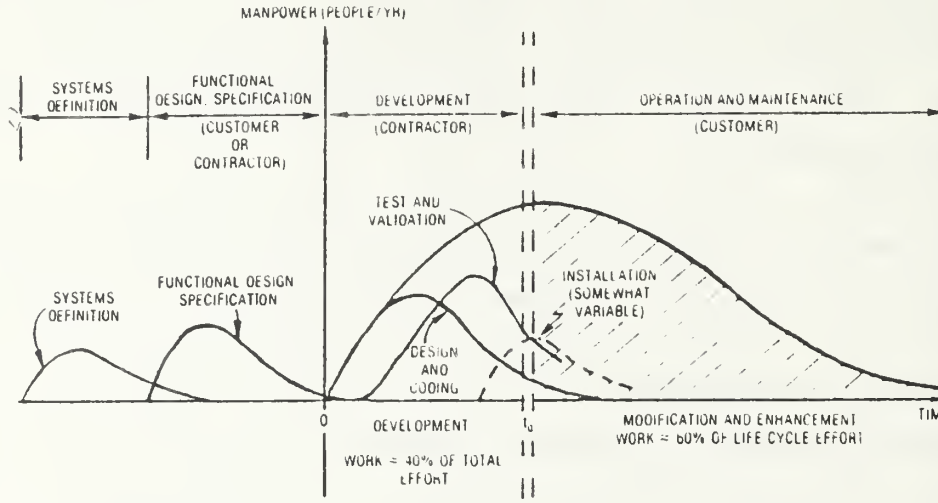
The goal of minimizing the impact of a modification on a software system is both complex and difficult to achieve. This is primarily due to the 'ripple effect'; the side effect of modifying software such that changes to one part of a software system affect other areas of the system [Ref. 11: p. 154]. The ripple effect is due to the various interrelationships between modules in a program and between

programs in a software system. Modules and programs may be related in the terms of functions or variables they share. Any change to a module has the potential to propagate its effect throughout the code. Changes to correct errors show at least a 20 - 50% chance of generating further errors [Ref. 12: p.122].

The effort and the difficulty of implementing the change is not simply a matter of rewriting the necessary code to implement the change, but must also include an examination of other parts of the system to determine if additional adjustments to compensate for the change must be made. Usually this involves a manual search of the code to identify any other affected modules, a process that often requires more time and effort than rewriting the code.

The software life cycle is represented graphically in terms of resource (usually manpower) use over time. There are several views as to how such a representation should look. One view, that of Putnam and others [Ref. 13], holds that the life cycle closely resembles a Rayleigh curve with the inflection point representing the delivery of the software system to the user and the start of the maintenance phase (Figure 2.3). The bulk of the effort occurs in the maintenance phase. The effort required to maintain a system steadily decreases over time [Ref. 13: p. 12]. Enhancements that exceed the level of effort should be treated as new development.

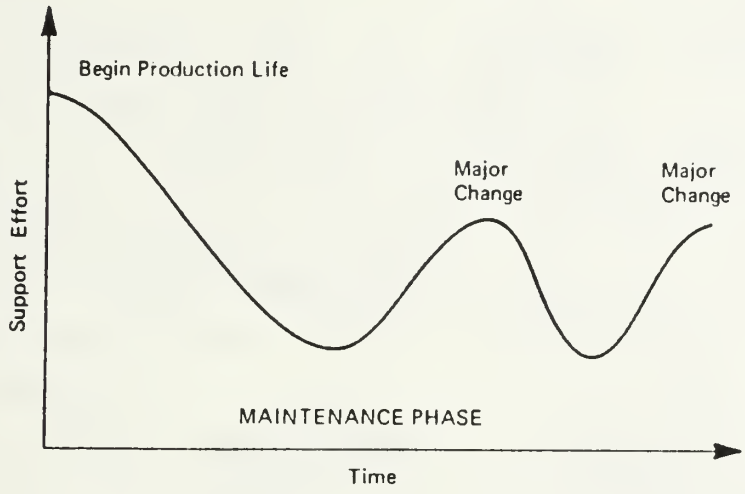
An alternative view holds that the effort varies over time as each new enhancement request initiates a mini-development cycle (Figure 2.4). All enhancements, regardless of scope, are treated as continuation of the original system instead of new developments. This view supports the software evolution perspective taken in this thesis, and seems to better represent the industry and government policy of issuing successive "releases" (major changes or revisions) of a software system.



[Ref. 13: p. 15]

Figure 2.3 Software Life Cycle - Putnam.

Ideally, given a stable maintenance environment working on a well-documented, well-designed system using maintenance techniques incorporating state-of-the-art technology, the curve in Figure 2.4 will gradually decrease. Each 'hump' will be lower than its predecesser as the system is gradually refined and the ripple effects are tightly controlled. Unfortunately, the reality is more accurately represented in Figure 2.5, where the resources required to support the system increase steadily over time. Enhancements are requested that exceed the capacity of the system to evolve. Poor design practices, poor documentation and poor maintenance practices fuel the ripple effect and errors propagate through the system. Any oscillation effect due to enhancement is dampened out in the continual battle against bugs.



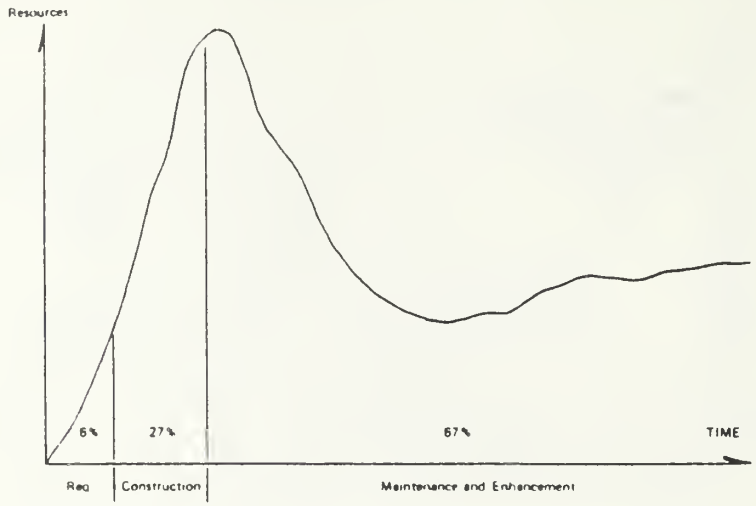
[Ref. 11: p. 122]

Figure 2.4 Software Life Cycle - McClure.

E. LAWS OF PROGRAM EVOLUTION AND MAINTENANCE

Studies by Belady, Lehman and others have shown that there exists a deterministic, measurable regularity in the life cycle of a software system. This regularity has been expressed in the five laws of large program evolution dynamics. These laws have been supplemented by Barry Boehm's three laws of software maintenance. These laws accurately represent observed phenomena in software evolution, and are useful to the software manager in understanding how and why software evolves.

1. Law of Continuing Change: A system that is used undergoes continuing change until it is judged more cost effective to replace the system with a re-developed version.
2. Law of Increasing Entropy: The entropy of a system (its unstructuredness) increases with time unless



[Ref. 14: p. 159]

Figure 2.5 Software Life Cycle - Reality.

specific effort is made to maintain or reduce it [Ref. 4: p. 2-3].

3. Fundamental Law of Program Evolution: Program evolution is subject to dynamics which make the programming process self-regulating, with statistically determinable trends and invariances, while appearing to be stochastic locally in time and space.
4. Law of Invariant Work Rate: The overall level of effort in a large programming project is statistically invariant, or tends to remain fairly constant over time.
5. Law of Conservation of Familiarity: For reliable, planned evolution, a large-program undergoing change must be released at regular intervals determined by a safe maximum release content. If the interval spacing or maximum content limitations are exceeded,

integration, quality and usage problems will occur with the resultant time and cost over-runs [Ref. 15: p.142].

Some additional laws of software maintenance were presented by Barry Boehm in [Ref. 16].

1. Law of Organizational Reflection (Conway's Law): Software products and the organizations they serve grow to reflect each other.
2. Law of Glacial Technology Transfer: Software products are rarely modified to accommodate a different technology.
3. Law of Maintenance Leverage: Organizational analysis and software design have more maintenance leverage than any development or maintenance actions.

III. FORCASTING MAINTENANCE

While software maintenance follows a cyclic pattern of progressive enhancements, it is generally performed as a level of effort activity [Ref. 1: pp. 545]. The cycles of enhancements and the difficulty of each enhancement are largely unpredictable very far into the future. The central problem cf forecasting software costs is predicting what the level of effort will be over the operational life of the software system.

Two primary factors influence the level of effort estimate. The first is the amount of software maintenance needed. Future software maintenance needs are driven by error repair and changes rising from external factors. Cperational systems fulfilling current and projected mission needs will naturally require maintenance for some time into the future, and may require considerable staff to support new releases and revisions. The second factor is the perceived benefit of the software to the organization, which depends upon the worth of the software relative to other resource requirements. The two forces combine to yield a level of effort sufficient to correct software errors and make most changes due to external factors within a reasonable time. It is not completely clear, however, how the amount of maintenance needed and its perceived benefit interact to determine a level of effort [Ref. 17: pp. 4].

The estimating problem is complicated by the unpredictable nature of maintenance ripple effects. Hopefully, analysis of available documentation and careful regression testing will help tc eliminate errors, but the software manager must recognize this complication to his estimation problem. While none of the published techniques or models

available to the authors specifically addressed the ripple effect in the estimating process, a realization of the phenomenon is often imbedded in the representation of maintenance personnel skill levels. Intuitively, the more experienced analysts and programmers are more likely to detect potential ripple effects.

Once the level of effort has been determined, software maintenance labor costs are relatively easy to estimate using the appropriate labor rates. Software maintenance is a labor-intensive activity, and labor costs are dominant. Costs associated with computer hardware and support software may be included, but such costs are normally attributed to activity overhead as those elements benefit other activities in addition to the maintenance of a particular software system. A software manager should be aware of the benefits of acquiring sophisticated support software to replace maintenance personnel [Ref. 18: p. 247].

The future need for software maintenance and its perceived benefit are difficult to quantify. Thus the software manager requires methods somewhat more quantifiable and sustainable to generate reasonable estimates of software maintenance costs. The following chapters will discuss such methods and how a software manager should approach the task of estimating the software maintenance level of effort.

The foundation of any approach to forecasting software maintenance is the estimator's own experience and judgement, the blend of which will hereafter be referred to as "experienced judgement". The software manager must apply his or her own experienced judgement to the forecasting/estimating methodology. Experienced judgement is either applied directly, as in direct estimating or estimating by analogy, or it is used to directly estimate the parameters upon which a parametric model is based. Published cost estimating models reduce the amount of judgement needed by providing

table of values for all parameters used, and the role of the estimator is reduced to one of picking numbers and plugging them into formulas. One must remember that those models were derived from the model designer's experience, modified by statistical analysis of sample populations, and will not apply to all environments. The software manager must understand and appreciate the characteristics and limitations of any approach used to forecast future software maintenance needs. Any approach the estimator cares to use will yield an estimate. The accuracy of that estimate depends upon the estimator's understanding of the software being maintained, the environment within which it will be maintained, and the applicability of the cost estimation approach to the software and the environment. The estimator must ask the following question: "Does this approach fit my situation and needs?"

IV. DATA REQUIRED FOR MAINTENANCE COST ESTIMATION

The basic management tenet: "You can't manage what you can't measure" applies to the management of software maintenance with the caveat "You can't measure what you don't keep data on." Accurate and complete data collection is the heart of any algorithmic technique to estimate software maintenance costs. Without good data, the parametric values of the model cannot be reliably derived and the model cannot be accurately calibrated to the maintenance activity environment.

The question is then raised "What data must be collected?" The data required falls into two broad categories:

- Characteristics of the Software
- Characteristics of the Maintenance Environment.

The characteristics presented in Table II and below are derived from published analysis of software cost estimation models [Ref. 19, 20], and the authors' own analysis of available models [Ref. 1, 13, 21, 22]. The listing is not all inclusive: the immaturity of software maintenance cost estimation is such that an attempt at presenting a comprehensive list of all variables that influence software maintenance would be presumptuous. It is intended more as a reference to the software manager in the hope that he or she may be guided toward a better understanding of the scope and nature of the task.

TABLE II

Software Cost Data Elements

Software Characteristics	Environment Characteristics
Development History manpower (MM/yr) total effort total time environmental descrip	Personnel experience language familiarity support S/W familiarity application familiarity participation in design personnel continuity real productivity
Maintenance History valid errors found enhancements started enhancements deferred emergency fixes started original LOC modified LOC new LOC original modules modified modules new modules total modules	Computer attributes size and age memory constraints machine constraints operating system access of maintenance personnel to computer scheduling priorities
Type of Program application language used structure	Software Tools software tools available to maintenance personnel
Complexity size operators operands degree of uniqueness algorithm complexity H/W - S/W interfaces input - output files module complexity	Programming Techniques Extent to which modern programming practices are used
Documentation top-level detail currency	Data Base size availability to maint personnel

LOC - lines of code
H/W - hardware
S/W - software
MM - man-months

A. SOFTWARE CHARACTERISTICS

1. Development History

Manpower during development (MM/yr): The man-months (MM/yr) per year of the the development phase, broken down by phase of the lifecycle (e.g., Requirements Analysis, Specification, Design, Coding, Testing) and by labor mix (e.g., programmers, analysts , documentation specialists, etc.).

Total development effort: The total number of man-months expended during development.

Development time: Calendar months of development.

Description of development environment: A description of the development environment to include

- computer used
- tools and automated programming aids used
- languages used
- software engineering techniques and modern programming practices used.

It should be noted which of the above were new to the development environment.

2. Maintenance History

No. of valid errors found per month: Valid errors found since program acceptance

No. of enhancements started per month: Number of user or environment driven enhancements started since program acceptance.

No. of enhancements deferred per month: Enhancements deferred for what-ever reason since program acceptance.

No. of emergency fixes started per month: Emergency fixes since program acceptance.

Original lines of code: Lines of code in program at acceptance.

Modified lines of code: Lines of code modified since acceptance.

New LOC: Lines of code added since acceptance.

Total LOC: Cumulative lines of code.

Original modules: Modules in program at acceptance.

Modified modules: Modules modified since acceptance

New modules: Modules added since acceptance.

Total modules: Cumulative modules in program.

3. Type of Program

Function: Scientific, transaction processing, real time control system, operating system, etc. The logical function has a significant impact on the complexity of the program.

Language used: High order language (HOL) --COBOL
FORTRAN, PL1, etc.

Assembly language

Uniqueness of language--is it common and well-known like FORTRAN or a specific, obscure assembly language?

Structure: Those attributes that contribute to the readability of the program form [Ref. 23: p. 72]. The hierarchical representation that indicates the relationship between modules. A subjective parameter value is useful here. "Well-structured" could mean code featuring independent modules employing parameter-passing and information hiding. "Poorly structured" could refer to spaghetti code replete with GO TO's .

4. Complexity

Size: Program size is measured in "lines of code", an expression which can mean many things. Exactly what constitutes a "line of code" is difficult to define because programs consist of more than executable statements. Programs may include comment lines, data declarations, job control language statements, format statements and macro-instructions. A counting method may consider every statement to be a line, whereas other methods may only consider a subset, such as executable lines and data declarations. Barry Boehm uses "delivered source instructions" as his vehicle, and defines it as follows:

> This term includes all program instructions created by project personnel and processed into machine code by some combination of preprocessors, compilers, and assemblers. It excludes comment cards and unmodified utility software. It includes job control language, format statements, and data declarations.

[Ref. 1: p. 59]

A more subtle problem occurs when counting lines of code for programs written in HOL. FORTRAN commonly uses one statement per line, although continuation lines are allowed and some FORTRAN versions allow multiple statements per line. A freely structured language like COBOL uses punctua-

tion to delimit statements and a line of code may contain several statements. A line of code in PL1 may be everything written between semicolons.

Recognition of the problem of how to measures size is necessary to effectively manage resources. Programmer productivity metrics are meaningless unless the software manager understands the line-counting rules in effect. These rules should be documented and clearly understood by all who interact with software maintenance.

Operators: The number of unique operators and the total number of operators in the program.

Operands: The number of unique operands and the total number of operators in the program. Operators and operands are used in M. Halstead's Complexity metrics [Ref. 24].

Degree of uniqueness: A subjective measure of the uniqueness of the function and the software system. The impact here is personal familiarity with the problem, the hardware and the software. The more common the function, hardware and software, the lesser the degree of complexity and the more likely maintenance personnel will quickly understand the system.

Complexity of algorithm: Again, this is a subjective measure. A more complex and sophisticated algorithm (e.g., electromagnetic signal analysis) will be more difficult to understand than a relatively simple one (e.g., payroll calculation). If the mathematical sophistication of the underlying algorithm is beyond the perspicacity of the programmers and analysts available then there evolves a strong inclination not to touch the program for fear it will "break".

H/W - S/W interfaces: Types of interfaces include data storage and retrieval devices, on-line communication devices, real-time command and control, and interactive terminals. The number and diversity of interfaces directly impacts the complexity of the system.

Input-output files format: The number of different formats the system reads and outputs, including card, tape, disk, or screen formats. The type of file format and the number of files accessed may impact system complexity [Ref. 21 :B-2]. The DoD Micro Estimating Model used to estimate development costs incorporates different file formats as input parameters, but weights each the same [Ref. 19: p. A-15]. This implies the impact on maintenance costs is either negligible, or too dependent upon specific equipment to incorporate in a general model.

Complexity of modules: Table III compares the subjective complexity ratings as a function of the type of operation to be primarily performed by the module [Ref. 1: p. 391]. While the ratings are designed to be incorporated into Barry W. Boehm's COCOMO model, they do assist the software manager in understanding some of the characteristics of a program that directly impact complexity.

Documentation: Documentation is essential to software maintenance. Maintenance personnel must be able to understand how and why a program operates in order to perform software maintenance. Documentation is the tool used to gain that understanding. While software documentation is a controversial subject, most software experts agree on the following:

1. Well-documented programs are easier to work with that undocumented programs, but incorrect documentation is far worse than none at all.

TABLE III

Module Complexity Rating vs Type of Module

Rating	Control Operations	Computational Operations	Device-Dependent Operations	Data Management Operations
Very low	Straightline code with a few nonnested SP operators: DOs, CASEs, IFTHENELSEs. Simple predicates	Evaluation of simple expressions: for example, $A = B+C*(D-E)$	Simple read, write statements with simple formats	Simple arrays in main memory
Low	Straight forward nesting of SP operators. Mostly simple predicates	Evaluation of moderate level expressions, for example, $D = SQRT(B**2-4.*A*C)$	No cognizance needed of particular processor or I/O device characteristics. I/O done at GET/PUT level. No cognizance of overlap	Single file subsetting with no data structure changes, no edits, no intermediate files
Nominal	Mostly simple nesting. Some intermodule control. Decision tables	Use of standard math and statistical routines. Basic matrix and vector operations	I/O processing includes device selection, status checking and error processing	Multifile input and single file output. Simple structural changes, simple edits
High	Highly nested SP operators with many compound predicates. Queue and stack control. Considerable intermodule control	Basic numerical analysis: multivariate interpolation, ordinary differential equations. Basic truncation, roundoff concerns	Operations at physical I/O level (physical storage address translations; seeks, reads, etc). Optimized I/O overlap	Special purpose subroutines activated by data stream contents. Complex data restructuring at record level
Very high	Reentrant and recursive coding. Fixed-priority interrupt handling	Difficult but structured NA: near-singular matrix equations, partial differential equations	Routines for interrupt diagnosis, servicing, masking. Communication line handling	A generalized, parameter-driven file structuring routine. File building, command processing, search optimization
Extra high	Multiple resource scheduling with dynamically changing priorities. Microcode-level control	Difficult and unstructured NA: highly accurate analysis of noisy, stochastic data	Device timing-dependent coding, microprogrammed operations	Highly coupled, dynamic relational structures. Natural language data management

[Ref. 1: p. 391]

2. Good documentation implies conciseness, consistency of style, and ease of update.
3. A program should be its own documentation: that is, a well-documented program should take advantage of the self-documenting facilities offered by the language and should have its documentation built intc the source code to the maximum extent practicable [Ref. 25: p.175].

Documentation takes many forms. Robert L. Glass [Ref. 23: p. 163] offers two categories of documentation of interest to the software manager: top-level software definition and detail-level software definition. Table IV describes the two categories in mcre detail. Additional categories may include user, test and operation documentation.

Currency/correctness of documentation: To be of any value, documentation must be both correct and current. Documentation that dces not accurately reflect the current state of a system is worse than none at all. Unfortunately, most system documentation resides in tomes that gather dust on shelves. Maintaining documentation is a task that everyone tries to avcid, yet must be done if the software system is to survive. Tools are available to aid in this task.

B. ENVIRONMENTAL CHARACTERISTICS

1. Personnel

The impact of personnel characteristics on software maintenance and the management of personnel to accomplish the software maintenance function will be discussed in Chapter VI. This section will define the terms used.

TABLE IV

Maintenance-Critical Documentation

I. Top-level software definition (document)
   a. Overall structure summary
   b. Overall database summary
   c. Design decision data
   d. Underlying philosophy
   e.. Midlevel structure(s)
   f. Midlevel data base(s)
   g. Index to listing

II. Detail level software definition (listing)
   a. Commentary for
      1. Detail structures
      2. Detail database
      3. Detail functions
      4. Implementation anomalies
   b. Readable names
   c. Structured, indented code

[Ref. 23: p. 163]

Programming experience: The number of years of programming experience an individual has. When used as an input to estimate cost estimating models, it is assumed that more experience has a positive impact on reducing costs. This may or may not be true, and is heavily dependent on the other characteristics listed below.

Familiarity with language: A subjective measure of the expertise an individual has with a particular language. Some studies have shown experience in a number of languages is of greater benefit than considerable experience in one language.

Familiarity with hardware and support software: A subjective measure of the experience an individual has with the computer and its support software (e.g., operating system, compilers and available software tools).

Familiarity with function: A subjective measure of the understanding an individual has of the software's function. This becomes important in complex functions, particularly so where the underlying algorithm is abstruse or the system is poorly-documented.

Participation in design effort: The degree to which an individual was involved in the design and development stage of the software. Such experience is invaluable in helping maintenance personnel understand the software's underlying logic and philosophy.

Personnel continuity: Personnel continuity may be represented as personnel turnover. A maintenance staff with low turnover will spend less time on job communication and training and more on productive work.

Real productivity: Productivity is a highly controversial metric that is extremely difficult to define. A typical productivity definition of "lines of code written per man-month" fails on four counts.

1. The definition of "lines of code" is imprecise, and a productivity measure incorporating it suffers from sensitivity to line counting variations.

2. "Man-month" is a measure of effort, not of productivity. While there is a correlation between effort and productivity, it can be represented using the metric "man-month" only with the greatest caution [Ref. 12: p. 16].
3. Coding is but a small part of the maintenance effort. The critical area of maintenance lies in understanding the program and what must be changed. There exists no acceptable metric for measuring the rate at which a human may understand a complex problem.
4. There exists a tendency to penalize HOL program in favor of assembly language programs when using a "lines of code" metric. Assembly languages require more lines of code to implement a given function than HOL, thus more lines of assembly code can be produced by a programmer during the coding portion of maintenance [Ref. 26: p. 41].

A more useful definition of programmer productivity may be in terms of programming functions per unit of time [Ref. 27: p.34].

2. Computer Attributes

Size and age: Physical attributes of the host computer that affect software maintenance include the size (eg. mainframe or mini), the age, memory constraints, machine constraints, and the operating system it will support. A large computer will support more sophisticated software tools than a smaller computer of the same age [Ref. 1: p. 460], and a new mini may have more capability than an older mainframe. The age of a computer is critical in terms of vendor support (enthusiasm to support a given architecture declines with time), processing capacity, memory and software sophistication.

Memory constraints: limitations are imposed on the performance of software maintenance by the size of the available memory. A machine whose production work consumes 90% of its memory leaves little to dedicate to enhancement maintenance.

Machine constraints: Machine constraints are the characteristics of a particular computer that may adversely impact software maintenance. These may include such characteristics as unique architecture, high operating costs or a machine-specific language version. Such constraints vary from activity to activity, but it is sufficient to say that a software manager should be aware of the limitations of the host computer.

Operating system: A sophisticated operating system enhances the productivity of maintenance personnel by allowing interactive testing and debugging. Turnaround time (the time between the entry of a command or a program and the computer's response) impacts the speed with which maintenance may progress. A sophisticated operating system that supports virtual memory and a wide range of software tools is far more conducive to effective maintenance than a batch-oriented operating system supporting a compiler.

Access of maintenance personnel to computer: The number of terminals dedicated to maintenance personnel, and the policies regarding terminal use.

Scheduling priorities: The priority given to maintenance functions. This is primarily a management concern, and requires both an awareness of and commitment to the importance of software maintenance.

3. Software Tools

Number and type of software tools that may be applied to software maintenance. Tools are discussed more fully in Chapter VII.

4. Programming Techniques and Standards

The extent to which modern programming practices (structured programming, information hiding, etc) are applied to software maintenance. Programming techniques are also discussed more fully in Chapter VII. Some measure of modern programming practices used are common to the majority of cost estimation models studied.

5. Data Base

The implications of data base to software maintenance are discussed in Chapter VIII.

C. RECOMMENDATIONS

While the data base required to estimate software maintenance costs often exists, the data are non-homogeneous. There are no definitive standard metrics; only a collection of interpretations. The definition of software maintenance itself may vary within an organization itself. A software manager who subscribes to the exclusive definition may be replaced by one who prefers the inclusive definition. Any data collected in the past would be of little value to the current manager. The definition of software maintenance also frequently varies from activity to activity. Additionally, the definitions of "lines of code" and "complexity" may vary from activity to activity. The data collected using interpretive metrics are generally unusable outside of its source environment.

To accurately estimate software maintenance costs, therefore, it is necessary to start with a standardized set of data. A standardized set of data must be collected using standard, universal metrics. It is hoped that DoD and industry may agree upon a uniform set of software metrics.

Once a standard set of software metrics for cost estimation is derived, data must be collected, stored in a centralized location, and applied to existing cost estimation models. The use of standard data would go far to improving the accuracy of current models. Analyses of the data may then be conducted that will result in the next generation of more precise, more accurate, and viable software cost estimating methodologies. A uniform data collection instrument must be designed that will enable data collection in a consistent manner. This approach is mandatory to avoid problems arising over which data to collect, when to collect it, and how to maintain the data in a machine readable format for storage and analysis.

V. MAINTENANCE COST ESTIMATION

A. OVERVIEW

The use of the term "art" to describe the process of estimating software maintenance costs is particularly apt. While much research has been devoted to software development cost estimation, little has been devoted to maintenance cost estimation. Indeed, until the Lientz and Swanson study [Ref. 2] the characteristics of software maintenance and the factors that influence it were imperfectly understood. Many techniques and parametric models exist today to estimate development costs but the few models available to estimate maintenance costs are simply extensions of existing development models, and generally assume that the same factors influencing development costs will also influence maintenance costs [Ref. 1: p. 536, 13: p. 7].

A broad distinction of approaches to estimating software maintenance costs include traditional methods and parametric models. Traditional methods rely primarily on the estimator's (or group of estimators') experienced judgement. Parametric models presume that relationships exist between costs and certain software characteristics [Ref. 17: p. 9].

B. TRADITIONAL METHODS

Direct Estimating

Direct Estimating is the application of experienced judgement in its purest form. The cost estimate is made based on the individual's knowledge, experience and judgement. Current knowledge and experience relative to the particular activity being estimated is vital to a creditable estimate. Excellent judgement is critical since future

maintenance activities are not apt to be the same as previous ones. Direct estimating may be combined with decomposition to yield a more accurate estimate. The software system may be decomposed into successivly lower functional subcomponents. When a low enough level is reached to estimate accurately, the estimator applies any appropriate technique to estimate each component's cost. Table V shows a possible subdivision of maintenance into functional subcomponents.

TABLE V

Software Maintenance Functions

Management/Supervision	planning, directing, coordinating, and controlling software maintenance activity
Administration	general office support
Analysis	studying a software problem prior to taking action
Design	developing a solution to a software problem
Programming	coding and unit testing of software changes
System Testing	formal testing of a changed software testing
Configuration Control	upkeep of master program libraries, backup tapes, program listings, etc
Documentation	making changes to user manuals, specifications, test plans, etc
Training	train users on program changes, training of new software maintenance personnel

[Ref. 17: p. 9]

The use of direct estimating and decomposition (essentially bottom-up estimating) offers the advantages of enhanced estimate quality since random error in the system estimate will be reduced by accumulating subcomponent estimates, and by enhancing the understanding of both the system and the maintenance task.

Analogy

Analogy is similar to direct estimating, and involves comparing the estimated effort of performing maintenance on a program with similar historical examples. The experience of another project serves as a baseline for the estimate, which is then modified by differences in project characteristics and available resources.

Judgement Enhancing Techniques

Judgement enhancing techniques are primarily based on experienced judgement. The accuracy of the estimate is enhanced through the use of methods that reduce the dependence upon one individual's judgements. These include Group Consensus or averaging. A group consensus technique may be a typical meeting, two individuals discussing the matter over lunch, or the more formal Delphi technique. The Wideband Delphi technique [Ref. 1: p. 335] seeks to improve the feedback of the Delphi technique and still avoid the pitfalls of group dynamics in a typical meeting. The process is time consuming, but

> ....has been highly successful in combining the free discussion advantages of the group meeting technique and the advantages of anonymous estimation of the standard Delphi technique [Ref. 1: p. 335].

A straightforward technique is to average several independent estimates. The independent estimates may be obtained using various estimating methods.

Traditional methods offer the software manager ease of use and familiarity of approach. Reasonable estimates of software maintenance costs may be obtained using traditional methods. However, the validity of the estimate remains dependent upon the ability of an individual (or group) to correctly analyze the past and make a valid judgement for the future. The analysis of the past may be affected by incomplete recall, biases, and inappropriate focus ("didn't see the forest for the trees"). The judgement of the future may be influenced by optimism, incomplete understanding of the existing system, or the pressure of deadlines and superiors.

C. PARAMETRIC MODELS

Parametric models presume that quantifiable relationships exist between software maintenance costs and certain software characteristics [Ref. 17: p. 9]. Such relationships are usually quantified by statistical analysis of historical software cost data. Once quantified, the relationships become variables that serve as major cost drivers in mathematical models.

Parametric models may take either a macro or a micro-level approach, or employ a combination of both. In a micro-level approach, the model addresses the individual components of a system. This approach offers the advantages of decomposition: reducing the system to components for which the level of effort may be easily estimated, and enhancing the software manager's understanding of the system. A macro-level approach focuses instead on the overall system and its interaction with the environment. A macro-level model is more apt to deal adequately with the effects of external factors, while a micro-level approach is likely to be more effective in estimating potential

maintenance ripple effects. Parametric models have been categorized in a number of ways by different studies [Ref. 1: pp. 329, 19: p. 4-11, 17: pp. 7-12]. The authors feel that categories based upon how the model itself was derived are of more value that ones based upon the characteristics of the model. Such a distinction should aid a software manager in deciding the applicability of a model to his or her own environment. Robert Thibodeau [Ref. 19: p. 4-11] presents the following categories:

Regression: A class of model structures whose design is based on the selection of the life cycle element of interest (e.g., life cycle effort, development effort, or coding effort) and a hypothesized relationship between the element and a number of selected inputs. The parameters of the hypothesized relationship are obtained by regression and the model becomes a single cost estimating relationship.

Heuristic: This model structure combines observation and interpretation with supposition. It is the formal representation of the subjective process of applying experience. Relationships among variables are stated without justification (e.g., cost per pound decreases with increasing size, development effort is related to type of application). Then subjective , semi-empirical, or empirical adjustments are made to the base estimate. Heuristic models combine a number of different estimating techniques.

Phenomenological: This type of model incorporates a concept that is explained in terms of a basic phenomenon that is not limited to the mechanics of software development.

Parametric models offer the software manager several advantages.

1. They are objective and not strongly influenced by personal biases or motivation.

2. They are repeatable, given the same input parameters.

3. They objectively represent historical cost experience and are calibrated by historical data.

4. They are efficient and able to support further estimates or sensitivity analysis.

5. They are easily automated.

6. Finally, they offer a supportable conclusion, one more likely to survive the scrutiny of budget-conscious superiors.

While parametric model are superior to traditional methods in most respects, they are not, however, perfect. Most models are not satisfactory for wide range of applications without considerable adjustment. The disadvantages of parametric models include:

1. Historical data used to derive and calibrate the model may not accuratedly represent the present or future. Research to date on software cost estimation has often been based on systems developed using outmoded, inefficient methods.

2. They are unable to deal with exceptional conditions.

3. Models cannot compensate for poor estimates of parametric values (garbage in - garbage out).

4. The majority of models available are either not applicable to the maintenance problem or represent it imperfectly [Ref. 1: p. 342].

Parametric models can be used to estimate software maintenance costs with reasonable accuracy. As with any tool, the tool user must fully understand how the tool operates and how to use it effectively. Effective use of parametric models to estimate software maintenance costs require understanding several key issues.

1. Every model is dependent upon experienced judgement for its parametric values. This is particularly true for subjective factors such as system complexity and experience of personnel. There is no realistic way to avoid using experienced judgement to estimate maintenance costs regardless of the method selected.
2. The model must be calibrated to one's own environment. The model itself is normally developed from a representative sample, as in Barry Boehm's COCOMO [Ref. 1], or from an observed phenomenon of software development, as in Lawrence Putnam's SLIM [Ref. 13]. Modifications of certain parameters must be made to "fit" the model to a particular environment. These modifications can be done either by the software manager or by an expert consultant with experience in the model. Either way, the calibration process is almost entirely judgement-dependent.
3. The software manager must have access to considerable historical data about the system being maintained and the maintenance environment. This data is critical to estimating parametric values and calibrating the model. Unfortunately, few software activities understand the importance of accurate records of software maintenance, nor are they aware of what characteristics of the software and of the environment should be monitored and recorded to support the cost estimation function. Data management and its relation to software maintenance is addressed in Chapter VIII, while a discussion of the characteristics of software and the environment that should be monitored and recorded was discussed in Chapter IV.

D. ESTIMATING MAINTENANCE COSTS

Several different software cost estimating models have been developed and used by DoD and industry, with varying results. This thesis will not evaluate any particular model. A summary of studies done to evaluate existing models is presented in [Ref. 28: p. 10]. Instead, this section will focus on considerations for planning an estimate, criteria to subjectively evaluate a software cost estimating model, and will summarize a view of the status of software cost estimation within DoD.

1. Planning an Estimate

Developing an accurate software maintenance cost estimate requires a significant amount of effort. The software manager should plan the estimate just as any project. The process for planning an estimate developed by Barry Boehm [Ref. 1: pp. 310-328] and tailored for software maintenance by G. Klemas [Ref. 17: pp. 30-31] is summarized in Table VI.

2. Evaluating a Software Maintenance Cost Model

How can a software manager evaluate the applicability of a particular model to his or her own environment. Barry W. Boehm offers the following criteria:

1. **Definition:** Has the model clearly defined the costs it is estimating, and the costs it is excluding?
2. **Fidelity:** Are the estimates close to the actual costs expended on the projects?
3. **Objectivity:** Does the model avoid allocating most of the software cost variance to poorly calibrated subjective factors (such as complexity)? Is it difficult to jigger the model to obtain any result you want?

TABLE VI

Software Maintenance Cost Estimating Procedure

1. Determine the purpose and objective of the estimate. Identify all costs that need to be included in the estimate and establish accuracy requirements.

2. Prepare the estimate plan, stating purpose, objectives and requirements of the estimate. Determine data and expertise needed to make the estimate, and decide upon a technique. Specify resources and time needed to make the estimate.

3. Review the plan. Verify objective validity and resource availability.

4. Gather the necessary data. This stage will be relatively direct for an existing system provided there is a current program maintenance manual. If an estimate needs to be done for a recently delivered system, obtain as much data as possible about its development. Compare with the best historical data available on similar systems.

5. Obtain several independent estimates using various models. Evaluate applicability of a model to estimating situation, and calibrate to own environment. Apply experienced judgement to independent estimates and derive an estimate that optimally satisfies the software maintenance requirements.

6. Verify that the estimate makes sense.

7. Document the verified estimate and stand ready to change it.

4. Constructiveness: Can a user tell why the model gives the estimate it does? Does it help the user understand the software job to be done?
5. Detail: Does the model easily accommodate the estimation of a software system consisting of a number of subsystems and units? Does it give accurate phase and activity breakdowns?
6. Stability: Do small difference in inputs produce small differences in output cost estimates?

7. Scope Does the model cover the class of software projects whose cost you need to estimate?
8. Ease of use: Are the model inputs and opticns easy to understand and specify?
9. Parsimony: Dces the model avoid the use of highly redundant factcrs, or factors which make no appreciable contribution to the results? [Ref. 1: p. 476]

E. DEPARTMENT OF DEFENSE AND SOFTWARE COST ESTIMATING

Department of Defense has a requirement for a software cost estimating model and methodology at three stages of the software lifecycle [Ref. 28: pp. D19-22].

1. Requirements Analysis: The objective here is to examine long-range ccsts of the software given a reasonable system proposal. Cost/risk assessments and budgetary estimates are performed here. Table VII shows the required inputs and outputs of such a model. The accuracy required by a model in the requirements analysis phase is less than or equal to 50%.

2. Specificatic and Design: A software cost estimating methodology can be used to assist the government or contractor in estimating the costs of a particular system design on either a near-term or longer-term lifecycle cost basis. The majority of the existing models take this perspective. Table VIII shows the required inputs and outputs cf such a model. Estimate accuracy required in this phase is within 25% of the actual.

3. Development, Cperations and Maintenance: A software cost estimating methodology can be used to assess the cost impact of changes during the development phase, and estimate the cost of implementing a change during the operations and maintenance phase. Table IX shows the required inputs and outputs of such a model. Required model estimate accuracy is within 10% of actual values.

TABLE VII

Model Parameters for Requirements Analysis Phase

INPUT DATA	OUTPUT DATA
System Requirements performance testing	Long Range Budget Projections system R & D prototypes production est. maintenance est.
Support Philosophy maintenance manning	
Technology hardware software	
Historical System Data Base similar systems similar technologies similar methodologies	

[Ref. 28: p. D21]

A major element of the DoD Software cost estimation goals is establishing a reasonable, representative and standardized methodology [Ref. 28: p. D23]. DoD should not adopt any specific model and declare it the standard; no model offers the accuracy required by DoD, nor does any model adequately represent each phase of the lifecycle [Ref. 19: p. 5-29, 28: p. 16]. Instead, DoD should

> ....specify the general procedure for estimating software costs (i.e., major activities, model selection, model documentation, estimate documentation and management actions required to use the results of any software cost estimation effort). The establishment of this estimating methodology should be in concert with the data collection goals and should make use of the data collected to "fine-tune" current models and develop new models. The model/methodology developed should possess the following attributes: [Ref. 28: p. 24]

TABLE VIII

Model Parameters in Specification and Design Phase

INPUT DATA	OUTPUT DATA
System Requirements performance testing	Long Range Budget Projection system R & D prototypes production est. maintenance est.
Documentation Reqmts MIL-STD commercial	Mgmt Support Activities trade-off analysis risk analysis resource mix impact assessment -schedule -personnel
Schedule Requirements	
Support Philosophy government contractor manning	
Cost Estimates system LCC hardware software actual vs predicted	Development Cost Estimates software hardware support maintenance documentation facilities manning
Technology hardware software	
Historical Data similar systems similar technologies similar methodologies	

[Ref. 28: p. D22]

- Open discipline: The methodology should be flexible and adaptable to specific environments. The procedure for selecting a specific model should allow for the exercise of discretion.
- The use of multiple models: The methodology should allow for the employment of a number of models as required by the lifecycle phase of for the application.

TABLE IX

Model Parameters in Development and Maintenance Phase

INPUT DATA	OUTPUT DATA
Schedule	Manhours by Function design code test documentation, etc.
Software Characteristics lines of code language complexity	Schedule
Testing Requirements	
Support Tools support software test software support personnel	
Test Facility Capabilities	

[Ref. 28: p. 24]

- Reproducible: The methodology used should yield the same estimate of cost given the same data and situation.
- Living methodology: The methodology must be updated constantly to reflect the current state of software technology. This is achieved through institutionalizing methodology and through DoD instructions, regulations and standards.

Desirable characteristics of a software maintenance cost model include:

1. Automated execution
2. Transportable for all commonly used computers. Written in HOL.

3. Model algorithms should be thoroughly documented and available to all users.
4. Outputs should be flexible and tailorable to several applications.
5. The total set of models should cover the whole software lifecycle, although individual models may be specific to certain phases or certain applications .
6. Models should deal effectively with missing data.
7. Models should be conservative of use of resources for data loading and computer time [Ref. 28: p. 26].

Desirable outputs of a model include:

1. Total manpower effort by phase and by effort type
2. Reasonable development time
3. Amount of documentation required
4. Staffing profile
5. Computer costs
6. Cost-schedule trade-off factors
7. Sensitivity of output to input variations
8. Expected maintenance required
9. Milestone occurance times
10. Risk Profile [Ref. 28: pp. 26-27]

F. THE DEATH OF SOFTWARE

An early objective of this thesis was to propose a model for predicting the point at which the software system must be replaced. That objective was beyond the scope of our effort. Instead, some views are offered on what to think about.

It has been demonstrated that there are no hard and fast rules that may be used to accurately predict the lifespan of a system. Many factors come into play, and the influence of any factor varies considerably from system to system.

In general, the lifespan of a system may be said to be over if:

•It fails to adapt to change

•It is replaced by another system performing the same function [Ref. 29: p. 32].

While replacement of software is conclusive and obvious, failure of software to adapt to change requires further explanation. Four primary changes [Ref. 29: pp. 32-33] may cause the death of a software system:

1. Hardware changes: Changes of this nature may be as catastrophic as the replacement of the entire computer system or as relatively simple as the expansion of peripherals. In either case, software systems written in machine-specific language may well be doomed. Even so-called "standard" languages like COBOL and FORTRAN are not immune, there being almost as many versions of these standard languages as there are computer manufacturers. Computer manufacturers recognize the difficulty of converting software systems and advertise compatibility between their product and a competitor's. The vendor's definition of compatibility and the user's may differ considerably, however.
2. Software changes: All software systems depend upon others. Applications programs depend upon other programs for input and the operating system for resource control. The operating system in turn, relies upon its compilers and utilities. A major source of software change is the manufacturer's system software, the package of operating system and associated utilities required to operate the computer system. A change to system software may have a

catastrophic impact on application software systems, but usually system changes occur incrementally. The application system must therefore adapt to each incremental change, or risk being rendered inoperable.

3. Changes in requirements: As previously noted, enhancements due to user requests are the major source of software maintenance. Many user requests result from a change in user requirements, often because the requirements were poorly thought out in the original design. If the original design or the software system's internal factors are such that modifications cannot be made to meet changes in requirements, the system falls into disuse and should be replaced with a system that can be evolved.
4. Changes caused by errors: All software contains errors. Correcting any single error usually introduces 0.5 further errors [Ref. 29: p. 33], so the error correction process never ends. Software that becomes riddled with errors is abandoned by users, and dies. Sufficient resources must be applied to the correction of errors to keep a given software system viable and healthy.

Given that software must change in order to survive, how can a manager economically justify any given change in the software? How does a manager know when to end the lifecycle of a software system and replace it with another? The answer is complex, and is influenced by economic factors, variable (and unknown) user requirements, rapid new technological advancements and other practical considerations.

The perceived benefit of the existing system can be thought of as the capabilities of the system and the value those capabilities have within the organization. This is clearly a subjective evaluation, and may be characterized as

the software manager's answer to the question: "What does this system do and what is the value of what it does to the organization?" The software manager must then compare the benefit of the system to the cost of operating the system. The second portion of the decision rule deals with comparing the existing system with the proposed replacement in two ways. First, the marginal cost of maintaining the existing system is compared to the marginal cost of implementing the proposed replacement. The method for comparing the costs of the two systems is probably best done using a marginal cost representation, such as the unit cost per transaction. Second, the perceived benefit of the existing system is compared to the perceived benefit of the proposed replacement. The replacement system must be at least as capable (i.e., equal perceived benefit) as the existing system.

Once the decision has been made to replace the existing system, the software manager must also decide the timing of the replacement. The benefit to the organization (in terms of capital and resources) of keeping the existing system for one more year should be compared to the additional capabilities expected from the proposed replacement if implemented this year.

The software manager's replacement decision rule may be stated as:

> If the perceived benefit of the existing system is exceeded by the cost of obtaining that benefit, and if the marginal cost of the existing system exceeds the marginal cost of the proposed replacement (including a factor for reliability problems with the new system), then the existing system should be replaced.

It is implied in the decision rule that there exits the opportunity cost of not having the use of the proposed replacement that must also be considered.

In the past, the major obstacle to replacing a system has been the considerable cost and uncertainty involved in developing software systems. This is true even today. The future holds promise, through the use of so-called fourth-generation languages and advanced software tools, of greatly reduced development cycles and considerably enhanced system reliability.

VI. PERSONNEL CONSIDERATIONS

A. INTRODUCTION

The DoD Joint Service Task Force Report on Software problems stated that "...people are the most important resource in any software or support effort" [Ref. 31: p. 24]. While the cost of hardware plummets, the cost of people is rising. By 1985 the cost of hardware will be at one-tenth the 1979 rate, and the cost of people will be at twice the 1979 rate [Ref. 32]. With manpower as the dominant element of cost in performing software maintenance, the software manager must better understand the critical aspects of personnel management in software maintenance. Considerable gains can be achieved through effective management of maintenance personnel and of the maintenance function. The personnel issue will be examined from two perspectives; that of skills and attributes are required in a maintenance programmer, and how to best organize maintenance personnel to accomplish the maintenance function.

B. SKILLS AND EXPERIENCE NEEDED IN SOFTWARE MAINTENANCE

The skills and experience required by the maintenance programmer are well summarized by a quote from the Pebbleman document.[1]

> To make this situation vivid, consider a navigation module on a supersonic aircraft. Let us suppose that the navigation module is supposed to provide the correct position of the aircraft to within 10 meters anywhere in

---

[1] Pebbleman is one of a series of Department of Defense (DoD) analysis papers which lead to the creation of the DoD standard language, Ada. Ada is a registered trademark of the U.S. Department of Defense [Ref. 33].

the atmosphere of the earth. The module obtains input from gyros, accelercmeters, clocks, doppler radars and navigation signal receptors which can listen to satellite and ground station signals. Suppose it has been determined (perhaps by exercise of self-diagnosing interface monitoring procedures and execution of fault-detection decision trees) that none of the input devices is [sic] malfuncticning. But suppose that the results produced by the module are consistently in error.

Let us further suppose that the actual error is a superimposition of errors from three separate sources: (1) a simple programming error involving unintentional clobbering of the contents of a global variable by a local procedure which incorrectly assumes that the global variable is local, (2) the decay in numerical accuracy of a certain class of computations through inadequate numerical analysis of error propagation, and (3) failure to design the module to take account of coriolis force, leading to systematic errors on north-south trajectories at high mach numbers.

Each of these error sources might fall within the province of distinct skills at the command of distinct trained specialists. Only a physicist familiar with the laws of kinematics and dynamics might be expected to realize and correct the coriolis force error. Only a numerical analyst familiar with the laws of numerical error propagation might be expected to discover and correct the error of numerical accuracy decay. And only a programmer trained in the use of nomenclature scope rules in the programming language used to implement the module might be expected to discover and correct the error of unintentional information clobbering.

If the actual error is a superimposition of these sorts of errors at these three sorts of levels of program logic, it is doubtful that a maintainer, trained only in one of the three relevant skills, could succeed in untangling the superimposed errors, in isolating their sources, and in making appropriate corrections.

In a similar vein, if the system is being enhanced to meet new requirements, the skills of requirement analysts and designers may be required to modify the requirements and the design incrementally and to bring the requirements and design documents up-to-date consistent with the enhancement. In fact, because of the presence of more constraints, incremental reanalysis and redesign might be more difficult than original analysis and design. It may not be enough for the maintainer skilled only in the implementation, test, and integration phases of the software life cycle to perform acts of enhancement that call for the replay of skills exercised by teams of skilled specialists at earlier life cycle phases -- teams now disbanded and unavailable. This is particularly likely to be true if the requirements and design levels of the system being enhanced demand skilled thinking in application domains widely separate from programming.

But we know that maintenance and enhancement may tend to occur under circumstances under which the original teams that performed the high level logic analysis and design (and which used special application domain skills remote from programming skills) have long since disbanded, leaving maintenance and enhancement tasks to those unskilled in the higher logic levels of the

system. Such maintenance circumstances are unpropitious unless techniques can be found to determine when to call in or recongregate teams of skilled specialists needed for fault detection, repair, or enhancement. [Ref. 7]

Thus, to summarize, a maintenance programmer must be a highly-skilled individual with the following qualities:

1. Skilled in the programming language used in the activity and well-versed in obscure features whose use by development personnel may hide subtle errors.
2. Knowledgeable in the function of the system and able to detect errors in logic.
3. Possess the keen, incisive mind of a detective who enjoys the challenge of sifting through obscure clues.
4. Possess all the skills required in software development, including those of the requirements analysts, system designers and technical writers (to update the documentation).
5. Be determined and optimistic.
6. Have a keen awareness of human psychology in order to understand the logic of the original development programmer.

Unfortunately, the maintenance programmer rarely embodies all these qualities. Normally, he or she is relatively inexperienced and new at the organization. Programmers were often started out in software maintenance to train them for the "real job" of software development. A programmer is thrust into working on old software systems running on obsolete equipment and managed in a crisis mode. The novice programmer learns to patch systems "to keep them running", gaining little job satisfaction and rarely seeing a job well done and completed as his counterpart in development would. As patches accumulate upon patches, the system gradually deteriorates.

A reason that software maintenance has become the home of the inexperienced and the ineffective lies in the poor connotation of the term "maintenance". The problem of management perception and the status of maintenance personnel was a serious point of discussion in the session on Management of Software Maintenance at the Software Maintenance Workshop, held at the Naval Postgraduate School, Monterey, California, December 6-8, 1983. Maintenance in the physical sense implies simply repairing the structure without making any real changes, something akin to scrapping the rust off a bridge. That is hardly the case in software maintenance. It has been shown that software maintenance is largely designing and implementing user-requested enhancements, an activity very similar to system development although lacking the advantages of a dedicated and trained development staff. The correction of failures, the "scraping off the rust", is only a small part of the total software maintenance picture. Software maintenance is a highly demanding and vital function, fully deserving of management recognition. Management must take steps to recognize the importance of software maintenance and enhance the status of the maintenance programmer.

Some psychological testing would seem to be appropriate to test the individual for some or all of these beneficial or hindering traits. Schneiderman highlights some of the tests in use, but also notes that our understanding of them is shallow [Ref. 34: pp. 57 - 62]. Some of the tests available include:

- Myers-Briggs Type Indicator (MBTI) which gives insight into the personality dimensions of the programmer of extroversion/introversion, sensing/intuition, thinking/feeling, judging/perception. The interactions of these pairings of traits is more important than the preference itself.

- Minnesota Multiphasic Personality Inventory (MMPI) which is used to determine information about the person's desire to please, honesty and candor.
- Strong Vocational Interest Blank (SVIB) which matches the individual's likes and dislikes with other members of specific professions.
- Computer Programming Aptitude Battery (CPAB) tests verbal meaning, mathematical reasoning, letter sense, number ability, and diagramming skill.
- Berger Test of Programming Proficiency (BTOPP) is designed to measure an individual's knowledge and proficiency in the basic principles and techniques of programming [Ref. 34: p. 61].

Validation and improvement of these and other tests are still needed.

The development and availability of personnel with the proper skills is no small matter. All personnel are confronted with the problem of maintaining currency in a rapidly changing technology. In the data processing community in general demand exceeds supply, but within the Department of Defense the problem has added dimensions [Ref. 31] that arise from the three areas where the personnel may be drawn, namely: the military, civil service or contractors.

1. Military

The service policy of rotating officers every two to three years reduces and disrupts the supply of qualified personnel. This is exacerbated by the Army and Navy policy of also rotating those officers trained in data processing into and out of assignments far removed from the computer field.

On the enlisted side the problems are intensified due to lucrative employment opportunities within industry. Once an individual is trained, the prospects of high-paying jobs on the outside are very good. A U. S. Air Force study [Ref. 35: pp. 1-5] revealed that the second term retention rate[2] is only about 50% for certain computer resource skills.

2. Civil Service

Although the Joint service report [Ref. 31] is directed at the entire life cycle of embedded computer systems, the problem of availability of skilled personnel is still the same for computer software maintenance in general. For the civil service work force, maintenance personnel must stay current in a number of closely related fields, including computer science and engineering, but the means to do so may be thwarted by government employment regulations.

> ....The personnel problem is exacerbated by the limitation of most entry level and middle technical/management civil service positions to the Engineering (GS-800) series in the Commands that acquire ECS [embedded computer software]. This excludes computer science and other related degree fields from pursuing careers or shifting to careers involving ECS acquisition. It should be noted that Civil Service regulations currently prohibit advertising a position as interdisciplinary when one of the disciplines is a "Professional" series (as is the GS-800 Engineering Series). [Ref. 31: p. 25]

From another report on maintenance in the commercial sector, Lientz gives a figure of 20-30% shortage nationally of systems personnel [Ref. 36: p. 9]. Lientz suggests that users may have to fill in this gap between the supply and demand of programming personnel, but that can only happen if advanced software tools, such as fourth generation

---

[2]A second term retention refers to an individual making a second obligation to military service after the completion of the first term of enlistment normally 4-6 years.

languages, are available to simplify the task. Tools are discussed in the next chapter.

3. Contractors

The problems with military and civil service labor often forces a heavy reliance on contractors. This dependency on contractors has problems of its own.

- The contract performance must still be monitored by someone knowledgeable in the field.
- Contractor personnel must be trained in the system. This may become counter-productive as turnovers within the contractor's organization occur over which the military manager has no control or when a contract is not renewed.
- The required competition for renewal of a contract and possible loss to another firm drains the corporate knowledge regarding the system.
- The use of contracted software creates long learning curves when training personnel to maintain any specific system.

C. PERSONNEL ATTRIBUTES

The specific interdependent personnel attributes required for the maintenance programmer go a long way toward forming the maintenance programmer in somewhat the same way as a development programmer, but with a twist. As has been discussed earlier, the familiarity with the application, the language and the hardware environment are still important, but in the case of the maintenance programmer for a different underlying reason. The maintainer is often called upon to fix a system in a crisis mode or try to deal with a

system that has little if any documentation. An ideal would be to have the programmer or just someone who participated in the design, available to respond when the documentation is inadequate (if there is even documentation at all). As can be seen, the maintenance programmer is a different kind of programmer with different productivity measures than can be advocated on the development side where the programming team approach is to produce "egoless" programming from a democratic group approach of a joint effort [Ref. 37]. This encourages the exchange of ideas and reduces the ownership of programs.

Glass suggests that the maintenance programmer will always remain the bastion of the individual worker [Ref. 23]. The individual certainly must respond to any number of applications with a detective's curiosity to find clues to the problem where they are not readily available. In maintenance work there is much more of an interface with the user creating immediate feedback and frequent rewards when the users are happy. Martin and McClure carry this further saying there is a place for the team approach still in maintenance programming [Ref. 25: pp. 429-435]. This approach can help the training of maintenance programmers as well as exchange of ideas on the various applications for which the group is responsible. A complement to the team approach is presented in [Ref. 38], and suggests that support personnel such as a librarian to monitor and maintain the documentation and a archivist to monitor and maintain file updates are needed. The "egoless" attitude of getting the opinion of another programmer on a problem or the implementation of a change should help to produce more error-free programming as well as being a good learning tool. One drawback still may be the size of the maintenance organization. A very small maintenance shop may not have as many opinions available to draw on though the attitude could still be there.

D. A MAINTENANCE PROGRAMMER PERSONALITY PROFILE

The result of all of the questions of the organization of the personnel, and who is available to do what, may leave hanging the identity of the individual involved in this activity. From the many references, it would seem that this individual must have good sound judgement, vast experience and technical expertise, the ability to identify the needs of the user, great understanding of existing software and technical versatility. But, why is this multi-talented individual made to be the inferior to the development programmer? The exact reasons don't need to be defined, but the concept has grown through a process of evolution partly as a result of the definition of the term "maintenance" programmer discussed in Chapter II.

Bronstein and Okamoto propose that there really are separate types of individuals that should be working in the development and maintenance areas [Ref. 39]. This break is to be on the balance between an individual's "communication styles". From [Ref. 39] Figure 6.1 shows the four different psychic functions that combine to produce profiles of individual's attitudes, assumptions and reactions that make one more appropriate for different types of jobs. A definition of the terms from Figure 6.1 are:

- Analyzer (thinker) places high value on facts and figures and is good on judging relationships of things; wants to be in control of work.
- Affiliator (feeler) places high value on personal relationships; is flexible and thought of as a supporter.
- Activator (sensor) places high value on the here and now; is assertive; and therefore, supports time constraints.

- Conceptualizer (intuitor) places a high value on knowing the nature of things in terms of their overall significance.

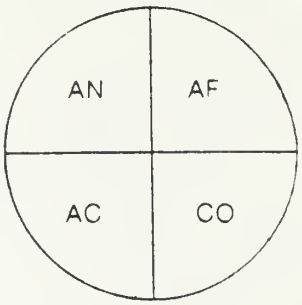
The level of each of these four functions may be questioned in relationship to the splits given in Figure 6.1, but there still points to the realization that according to one's communication style, a programmer may be more suited for the maintenance environment as opposed to the development environment. Finding the individuals who are motivated and best suited for this type of work will aid the manager in having competent and productive employees.

The detailed example given at the beginning of this chapter applies to the military tactical side of programming, but the variety of problems that any maintenance programmer will have to face will also cover the whole gamut of activities of that specific organization. Another example may be that of a space surveillance organization which could involve the fields of orbital and space physics, high level mathematics, intelligence processing, communications, etc. as well as data processing; a supply organization could involve inventory control, budgeting and financial management, accounting, purchasing, etc. along with data processing.

Some solutions can be seen in both getting better people in these positions as well as giving them better tools, environment and prestige in the work place.

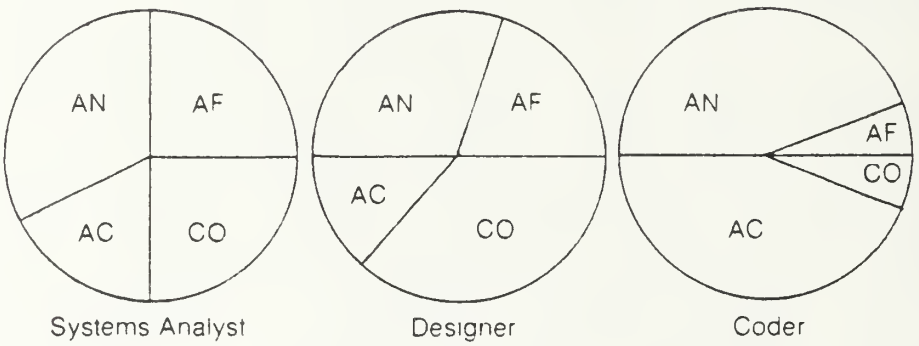
E. ORGANIZATION

There has long been a discussion of the organization of the personnel involved in the various programming activities. This is as shown in the 1972 discussion in [Ref. 40] of whether to have a separate programming organization devoted to maintenance entirely separate from the group

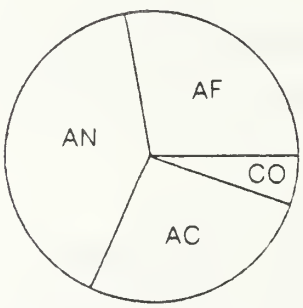


AN-Analyzer
AF-Affiliator
AC-Activator
CO-Conceptualizer

A salesperson must be able to communicate effectively in all styles. His style profile might look like this.



People good at various phases of programming might have profiles like these.



An effective and satisfied maintenance programmer often has this communication style profile.

Figure 6.1 Communication Styles.

devoted to the develcpment prccess. The skills needed in both these areas provide no definitive justification for either approach. When the same group of programmers is involved in both the design and maintenance of systems, there is a lot of cross training going on which will make any change easier to implement.

On the one hand, current research into some solutions to the 'Software Crisis' [Ref. 3, 11, 33] has concentrated in finding better ways tc accomplish and manage the development process. Methodologies such as Software Requirements Engineering Methodology (SREM) developed for the U.S. Army's Ballistic Missile program [Ref. 41] and Systems Analysis and Design Technique (SADT)[3] [Ref. 42] are comparatively new in this area and have aided in providing an orderly approach to the development process. These methodologies of SREM, SADT, and others combine methods and tools with human factors to aid in accomplishing the development process, such as to decompcse the software into modules, provide a graphical notation and control guidelines, sometime with the aid of computer software system [Ref. 43]. On the other hand very little of this has been done in the maintenance arena. This area is cnly now getting the attention it deserves.

---

[3]SADT is a trademark of Softech, Inc.

VII. TOOLS AND STANDARDS

A. INTRODUCTION

The resource of personnel is a dominant factor influencing software evolution. In the face of personnel demand outstriping supply, the software maintenance manager must obtain the maximum benefit from available resources. A means of achieving this is through the integration of software tools into the maintenance effort. A software tool is an automated program or process that enhances or replaces human effort. In Chapter III, Figure 2.1, it is pointed out that the bulk of a maintenance programmer's time (nearly 50%) is spent in trying to understand the existing software system. Thus, tools that can aid the programmer in understanding software should be addressed first by the software maintenance manager. Testing to maintain the integrity of the system is also a large part of this process, which can also be aided through the use of automated tools. The following discussion relates the availability and use of tools for the maintenance environment where it can improve programmer productivity.

B. SYSTEM VIEW

A more thorough view of the relationship between these activities and the tools available is in order, while still considering the personnel issues addressed in the last chapter. The tools addressed here are for the most part automated tools. While most of these tools discussed were created for the development environment rather than maintenance, they are still very applicable to the maintenance programming function. A DoD report [Ref. 31: p. A-39]

purports that in a total view of the system, there can be a dramatic impact on software problems achieved through support tools having five broad objectives. These objectives are:

1. Integration

This is designed to provide an interface with the entire environment viewed as cooperating functions.

2. Support

This brings the entire life cycle of the software together especially implementing and validating the changes after a system is designated operational.

3. Standardization

In this rapidly expanding world of computers, standards are designed for ease of transportation across a number of host processors.

4. Support of Standard Languages

Within DoD or any specific organization the designated standard languages must be supported by tools. In other words, completely language and machine portable support tools are not required.

5. Flexibility and Maintainability

The tool itself must also be flexible and maintainable within the environment to ease the evolutionary changes.

C. TOOLS

Software tools must therefore match the organization within which they are to be found. This is a broad statement addressing the large variety of sizes of data

processing organizations that can be found even within the authors' own experience within DoD. One may be dealing with software written in assembler languages on out-dated hardware all the way up to systems provided in the latest languages using state-of-the-art hardware and software technologies.

To avoid overly emphasizing either end of this spectrum, the authors are presenting some broad views of tools that may be helpful avoiding too much detail at either end of this spectrum. The availability of some specific tools that may meet the needs of a specific environment are addressed by type and vendor in a table in [Ref. 25: pp. 406-409] or in [Ref. 4: pp. 4-2 - 4-10]. A more current discussion of the type and availability of tools may be found in the numerous trade journals, a preferred source in this rapidly changing computer world. A list of sources to date can be found in appendix A. A comprehensive list of software tools would not be feasible nor desirable, as it would shortly be obsolete.

These tools though can help deal with past programming styles. This is not a criticism of past practices but an understanding of some of the problems facing the maintainer today. These include from [Ref. 39] :

- Maintaining programs written without standards.
- Lack of documentation and source.
- Different computers and languages.

D. TYPES OF TOOLS

Candidate areas for types of automated tools in a specific organization are suggested in the DoD Report [Ref. 31], and the Martin and McClure book [Ref. 25]. They may be categorized as:

1. Software documentation, such as structure charts, flowcharts and cross-referencing.
2. Testing and debugging tools.
3. Software libraries.
4. High order languages (HOL).
5. Configuration management.
6. Data base management systems.
7. Management information systems.
8. Analysis tools, such as simulation and diagnostic aids.

A rule of thumb for the manager may be to step through this list of types of tools that may be available and may be applicable to the specific organization. Certain old hardware configurations may have few choices of actual tools that are available. In the same light, old operating systems or software languages may not be supported in some areas. In any case items 1 to 4 can aid specifically in improving the maintenance programmer's understanding of the existing system.

The wide variations in specific functions that may also be addressed are shown in Table X from the National Bureau of Standards Special Publication 500-74, "Features of Software Development Tools" reproduced in [Ref. 44]. Suggestions for the development of new advanced tools that may overcome some of the problem areas mentioned are presented in [Ref. 6].

Table XI from [Ref. 25: p. 411] is presented to show a relationship between the types of tools available and the quality characteristics of the software. An emphasis within the organization for specific areas of improvements will force a manager to actively seek out one or more types of tools. Some examples of these types of commercially available tools are:

- static analyzer - Amdahl's MAP

TABLE X

Tool Function Taxonomy

Transformation	Static Analysis	Dynamic Analysis
Editing	Auditing	Assertion Checking
Formatting	Comparison	Constraint Evaluation
Instrumentation	Complexity Measurement	Coverage Analysis
Optimization	Completeness Checking	Resource Utilization
Restructuring	Consistency Checking	Simulation
Translation	Cost Estimation	Symbolic Execution
	Cross Reference	Timing
	Data Flow Analysis	Tracing
	Error Checking	Tuning
	Interface Analysis	
	Management	
	Resource Estimation	
	Scanning	
	Scheduling	
	Statistical Analysis	
	Structure Checking	
	Tracking	
	Type Analysis	
	Units Analysis	

- structure checker - TRW's CODE AUDITOR
- cross reference listers - TRW's DEPCHT, DPNDCY and FREF
- automatic documenter - General Research Corp.'s RXVP
- automatic flowcharter - TRW's FLOWGEN
- structuring engine - Catalyst Corp.'s COBOL Engine
- executive and performance monitor - TRW's PPE

TABLE XI

Software Quality Measurement Tools

Quality Characteristic	Measurement Tool
1. Understandability*	Structure checker Automatic flowcharter Execution path tracers Automatic complexity analyzer
2. Reliability	Execution path tracer Automatic complexity analyzer
3. Testability*	Automatic flowcharter Execution path tracer Automatic complexity analyzer
4. Modifiability*	Automatic complexity analyzer
5. Portability	Standard-language-version compiler Structure checker
6. Efficiency	Structure checker Performance monitor

* Defined as requirements for maintainability

E. ENVIRONMENTS

A list of the different types of tools that may be needed within an organization is a good start for the manager. The manager may then develop a list of those tools that are available for a specific environment, namely the computer hardware in use, the software languages being used, the database systems available, etc. These two lists may not overlap at all, and what's more, the tools that are available may not work with each other. For this reason, there is developing a strong emphasis on making available an environment that includes the tools needed for the computer language in use and compatibility with a variety of hardware manufacturers. Two environments under development are

specified here with some problem areas discussed. One is termed 'Program Manager' and the other is tied to the DoD language Ada.

One of the greatest problems within DoD is the use of obsolete hardware for which no tools exist. The question then becomes whether it is cost effective to retrofit the new tool to the old hardware or not. Unfortunately, the answer is usually no, but the question must be answered on a case-by-case basis.

1. Programming Manager

Dean and McCune [Ref. 6] and others state a need for a maintenance programming environment. A programming manager could be an integrated tool that would help improve the program development and maintenance process by ensuring the systematic application of managerial and technical policies and methodologies. There are three particular problem areas.

a. Standards and Policy

Management policies and standards are designed to promote quality and reliability of the software as well as minimize the retraining required. Unfortunately, the volume and complexity of the standards and policy are such that they are often ignored. Policy should be clear, direct and brief. Standards should be logically organized, indexed and useable.

b. Systems Details

As a programmer is working on a large system, a lot of time is spent learning how the system works. The programmer learns the minute details of how the system works through the process of modifying and debugging, but then promptly forgets this detail as work goes on into another

project. The usual methods of recording information in manuals, reports and memos is often not appropriate for this low level of information yet it is still vitally important to the maintenance function.

c. Programming Environment

Most programming environments have a number of tools available for use. Some of these are absolutely necessary and familiar to the programmer, such as editors and compilers. A variety of other tools may be available, but not well-known to the programmers. Manuals and on-line documentation provide very little help in this case since the programmer must explicitly request the tool. If the programmer has forgotten or doesn't know about them, they will remain unused.

2. Ada Programming Support Environment

Booch describes in [Ref. 33] the specific environment being developed for the new DoD language, Ada. This environment is referred to as the Ada Programming Support Environment (APSE). The Ada language and environment now being developed within the DoD is required for embedded computer systems only, thus far. (In the non-embedded systems there still is a need for some sort of program environment unless Ada becomes workable for both.)

The Ada Programming Support Environment seeks to support the system through its whole life cycle with the expectations from [Ref. 45] of:

- reducing compiler development costs
- reduced tool development costs
- improve software portability
- improve programmer portability

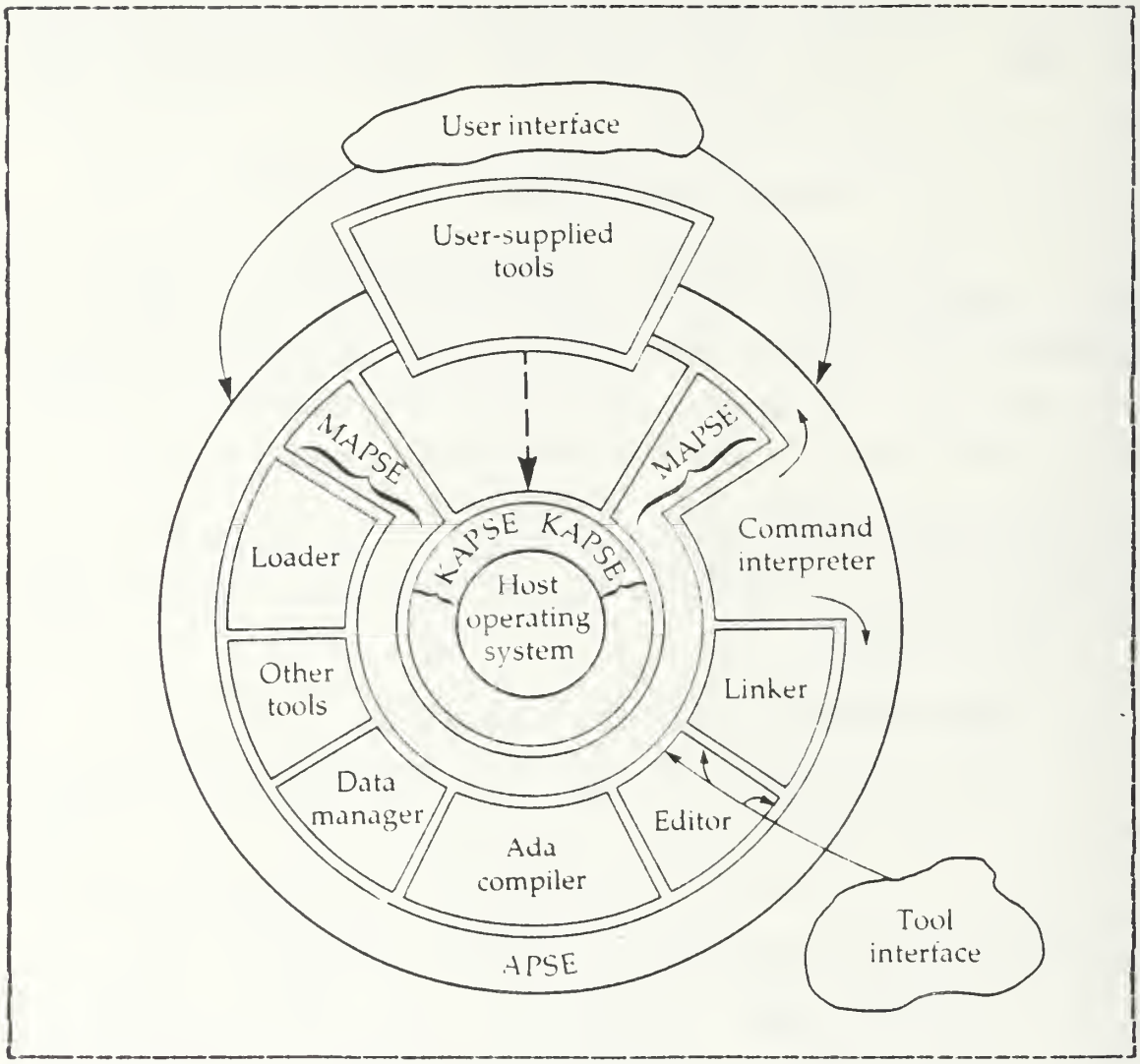


Figure 7.1 The Ada Programming Support Environment.

The architecture of the Ada Programming Support Environment is shown in Figure 7.1 from [Ref. 46]. The central point of control is provided for the project manager through the program data base. The data base physically exists at the inner level of the environment in the Host Operating system.

a. KAPSE

The Kernel Ada Programming Support Environment (KAPSE) is the next level which provides the logical to physical mapping for the APSE. This provides the most elementary requirements for run-time support. This support of the logical/physical mapping is the needed portability for the program. Theoretically then, the KAPSE would be the only implementation-dependent change needed for rehosting an environment.

b. MAPSE

Above the KAPSE is the Minimal Ada Programming Support Environment (MAPSE) which contains a minimal set of tools for program development, and, of course, maintenance. As defined by STONEMAN [Ref. 47], the MAPSE contains suggested tools such as:

- text editor
- pretty printer (code formater)
- compiler
- linker
- set-use static analyzer
- dynamic analysis tools
- terminal interface routines
- file administrator
- command interpreter
- configuration manager

c. APSE

The highest level and broadest view is the APSE itself. This includes a set of advanced tools to support all phases of the life cycle. Again STONEMAN [Ref. 47] does not specify specific tools, but does require tools for:

- creation of data base objects
- modification
- analysis
- transformation
- display
- execution
- maintenance (emphasis added)

F. USE OF TOOLS AND STANDARDS

The final question to the manager regarding the use of tools and standards within a specific organization is how they may be integrated to manage the function of maintenance. Gilb presents a possible way to organize these tools [Ref. 48]. He addresses one individual project, but the authors feel that the manager may use this system to evaluate a specific project or the organization as a whole. The process goes through a series of tables that are designed to determine what new tools (referred to as techniques) that the manager should actively seek out. Gilb steps through a simple project to demonstrate a manager's process of evaluation of one's objectives, priorities (referred to as quotas), and techniques already available within the organization. Some calculations between the organization's priorities and currently available tools demonstrate areas where the manager might want to actively seek new tools.

There is one caution in this area though from the DoD Joint Service Task Force Report [Ref. 31]. No widely accepted productivity measure exists for the various tools nor combinations of tools. Using tools with which maintenance personnel are familiar may be the most efficient utilization of personnel resources because it reduces mechanical activities and allows creative ones, but should not be limited to these when additional tools would be useful.

A standard emphasizes where personnel need to be trained. An example of a standards policy is can be shown within the Department of the Navy. A Navy instruction, SECNAVINST 5230.8, Information Processing Standards for Computers (IPSC) Program gives the overall policy information on high order language (HOL) standards, while attempting to avoid the proliferation of local- or vendor-unique standards. The objective is to identify, develop and implement standards that will:

- Provide for the greatest degree of compatibility between non-tactical ADP systems and their associated data systems.
- Facilitate the development of machine independent software.
- Provide for efficient operation and utilization of the ADP equipment.
- Incorporate and make available for general use related standardization efforts of individual ADP organizations.
- Increase reliability and transportability of software and facilitate backup and/or contingency processing.

A more specific standard, MAPTIS[4] High Order Language (HOL) Standard (OPNAV P160-S7-84) [Ref. 49], while recognizing the wide variety of unique problems to be faced within an organization as large as the Navy, further sets approval/nonapproval status for the MAPTIS program on the use of software languages. The latest language tools are divided into fourth generation languages, non-procedural languages and query/retrieval languages. An example of currently available data management languages is shown in Table XII from this standard. (This table is not intended to be a comprehensive list.) According to this standard [Ref. 49: p. 4], it is not intended to discourage the use of languages other than the already approved COBOL, FORTRAN, BASIC and Ada. Instead, it should force commands to demonstrate the cost effectiveness of a new language in the specific situation and to provide higher level authority with information about what and where languages are being used and to provide information for evaluating similar languages.

---

[4]MAPTIS is an acronym for Manpower, Personnel and Training Information Systems.

TABLE XII

Approved and Nonapproved Data Management Languages

LANGUAGE	VENDOR	DBMS	STATUS
Query/Retrieval Languages:			
OLQ	Cullinet	IDMS/R	A
CULPRIT	Cullinet	IDMS/R	A
T-ASK	Cincom	TOTAL	A
I.Q.	Cincom	TIS	A
COMPREHENSIVE RETRIEVAL	Cincom	TIS	A
DATAREPORTER	ADR	DATACOM/DB	A
DATAQUERY	ADR	DATACOM/DB	A
SQL	IBM	SQL/DS	A
SQL Plus	Oracle Corp.	ORACLE	A
DATASCRIPT+	Software AG	ADABAS	A
EASYTRIEVE	Pansophic		A
Nonprocedural Languages:			
FOCUS	Information Builders		N
INFO	Henco		N
MAPPER	Sperry		N
RAMIS II	Mathmatica Products		A
Fourth Generation Languages:			
MANTIS	Cincom	TIS	N
ADS Online/Batch	Cullinet	IDMS/R	N
NATURAL	Software AG	ADABAS	N
IDEAL	ADR	DATACOM/DB	N

A - Approved for use without waiver.
N - Not approved for use; requires waiver (if development effort is greater than 6 worker-months).

VIII. DATA EVOLUTION

A. DATA AS A TOOL

The area of data usage has two separate implications for software maintenance. First, there is the question of how the separation of the data from the application program effects the function of those assigned to "maintain" or improve, keep up-to-date, etc. the software system. The second implication lies within the research and development of the software tool called a data base management system (DBMS). Many tools and methods are being developed that can aid in the process and management of the software maintenance function. This can be just one of them.

In this day and age of the computer, most organizations are beginning to realize that no matter what the function of the organization (anything from product manufacturing to service-oriented financing), the information needed at all levels is an important resource. This has created the distinction between data and information. There is much raw data available, but information is that data which is put into a useable, correct, relevant and manageable form. Raw data is useless until it is formatted and made available. Correct and relevant information is needed at all levels. It becomes just as important for the supervisor in a bank operation to know the status of the transactions as it is for the bank president to know the cost and economies of the operations of the total organization.

The format of this information might be in any form from a logically organized file drawer to a computer system with automatic or query-driven, retrievable information. With more and more data being processed by any organization and

computer hardware technology costing less and less, the only cost effective way to process data of a very sizeable amount, is to process it on the computer. This may mean using computers from very large mainframes to micro processors or any combinations in between. There will not be too much distinction between the size of these computer systems placed here, since the same principles still apply, though sometimes to a lesser degree. The decision making process requires accurate and timely information. In the opinion of the authors it is becoming increasingly apparent then that the individual who controls the information is in control of the organization. Thus, we as a society are rapidly moving from the Industrial Age.

B. USE OF DATA BASE MANAGEMENT SYSTEMS

The need then to manage and control this data separately and effectively within an organization has created a data base environment. The DBMS itself can effect the success of the total package of maintenance tools. As Donahoe and Swearingen state: "....database is an essential requirement for configuration management and for using automated tools to maintain software" [Ref. 4: p. 5-2]. It provides a convienient means of storing test cases, providing error history and statistics, and cataloguing the detail program characteristics. The data base environment has also helped to get a handle on reducing some of the long-term maintenance problems and costs.

The data base environment has not always existed. It has grown from the recognition of the problems with the management of data. Analysis had shown that data should be handled separately from the functions that the software must perform. Today there exist many levels of this separation of data from functions in the various computer environments.

This development represents a change not only in software but in data processing management. The separation of data from function has evolved along with the other changes in the data processing field, such as hardware improvements and software languages. The separation has created the data base environment, where various programs and groups can have access to the same data and which, if properly implemented, can aid in the maintenance function. Martin and McClure [Ref. 25] have presented this separation as a progression through a series of four classes:

- Files
- Application Data Base
- Subject Data Base
- Information Systems

Martin and McClure have specified only these four classes of environments, but the authors feel that a fifth class for the distributed data base should be added. Each level increases the implementation complexity of the system, but adds to the management capability to handle greater and more diverse amounts of data. These five levels are defined below in a chronological order, but this is not necessarily implying that there must be a chronological movement (classes I to V) of the structure of data at a specific location, but rather, that the various combinations of these environments can and do exist at any one time within a single organization.

1. Class I Environment: Files

All early computer systems handled data operations as a file system. Systems were created to accomplish a specific function and the data description used was embedded

within the system. The problem was that an organization was not static nor was (or is) the data being processed. As more and more systems were automated, major problems were created. A high level of redundancy of the data was propagating throughout these different systems, creating difficult maintenance problems of data consistency and integrity. An apparently simple change could propagate a chain reaction of problems. These systems became very inflexible, especially when considering one time requests. File systems also were very expensive to maintain [Ref. 25: p. 118]. Often the great amount of money invested in existing file systems and the normal resistance to change have delayed the movement to the next level of environment. These costs are sunk costs though and should not be considered since they have no effect on the improvements or the maintenance of the system. Examples of file systems are VSAM and RMS.

2. Class II Environment: Application Data Base

The problems of the data changing while the function stayed the same created the need for a data base system to help manage and separate the data changes. The "data base" term is used in many forms of literature, but it is often only the currently popular term for a file system. A good definition from Martin and McClure is

> ....a shared collection of interrelated data designed to meet the needs of multiple types of end users. It can be defined as a collection of data from which many different end user views can be derived [Ref. 25: p. 117].

In any case the key is the storage independence of the data from the application programs plus the different logical views allowed of the data. Any modification of only the data then can be controlled independently. The class II environment was created quite naturally from the

process-oriented design. Systems each started using a data base, but each application created its own data base. This was easier to implement than the next level, but also continued almost all the same problems as class I environment with a high redundancy of data that would continue to proliferate as new functions were added. In addition to the high cost of buying this DBMS package, there would be the continuing high cost of maintaining the data. This pointed up the necessity for a Data Administrator (DA) or Data Base Administrator (DBA) to aid in the planning and control of this organizational resource. Some examples of the commercial packages are TOTAL by Cincom and IDMS by Cullinet, which originally came out in the early 1970's. The packages purchased for use in this environment could also be the same ones as those purchased for use in the next class III environment.

3. Class III Environment: Subject Data Bases

In this environment there is an actual design of the data structure done independently of the functions that must be carried out through the programming systems. Although this is the second environment to use data bases, it is the first to actually help reduce maintenance costs. An overhead is the initial time required to do the analysis and modeling of the data requirements, but this can reduce the time and cost later on in both the development and maintenance of application systems and their interaction through a single data base. This environment not only requires a change in the traditional analysis methods, but also in the traditional overall data processing management. Ideally, there would be active use of some sort of DBA to maintain planning and operational control of the data, but there must also be upper management support for this change in organization. If that is not done, an energetically started class

III environment can quickly degenerate into a class II environment [Ref. 25: p. 123].

4. Class IV Environment: Information Systems

This fourth class of data base is organized for the purpose of fast retrieval of information rather than the high volume production runs, which can work best in a batch mode. Some examples of these systems might be IBM's STAIRS or some of the relational models such as SQL and NOMAD, which also provide good query facilities for these user-driven systems. These systems are not difficult to implement and provide great flexibility for systems that require fast retrevial capabilities. On the other hand, they may not be efficient for systems requiring high volume transaction processing.

In a case where both retrieval and production runs are needed, trade-offs must be made between the two opposing requirements. This may be done through a combination of class III and class IV data bases where data is passed through an "extractor" program [Ref. 25: p. 127]. This would create two separate data bases where each is efficient for its specific function, but data also must be controlled and passed between the data bases on some schedule. This schedule may be on one or many possible conditions: online, offline, triggered by an event, ad hoc or even real time. Careful attention must be paid to ensure the integrity of both systems and the timing of each process. The major problem is that if both data bases are not locked from external use as updates are applied to both files simultaneously, the data bases could both become only partially accurate. An alternative approach might be to maintain a single data base and choose a system that was less efficient for either individual function, retrevial or production, but adequate for both. This may be done by using multiple

indexes or an inverted list. Only a thorough evaluation of the individual situation can determine the best trade-off.

5. Class V Environment: Distributed Data Base

In this age of the merging of the technologies of computers and communications, another environment for computer data most definitely is the use of data distributed throughout a computer network. Data used at one specific installation is handled through one or more of the classes I through IV. Data can be passed as files from one computer system to another, as needed. In the case of on organization whose functions are distributed among widely separate geographic locations, pieces of data are contained at these separate sites with a need for it to be managed by a single system. A network data manager has been proposed for this by a CODASYL committee to be another layer of their DBMS. Its extension would be called Network Data Base Management System (NDBMS). This would be another type of option that could be implemented on the DBMS that would manage the data resource requested on distant systems. The major drawback for this CODASYL NDBMS is that it requires a homogenous computer system.

Another and more well-known attempt in this direction is the System for Distributed Data Bases (SDD-1) by Computer Corporation of America. This system was designed for the Department of Defense's ARPANET. It is designed to handle the problems in relation to a global data directory, conflicts with possible deadlocks, and problems of efficiency. The replication of data at different sites is permitted, if it is determined that duplication is more efficient than the transmission costs involved [Ref. 50].

Either of these systems allows a choice for the organization that has many types of data and a requirement to access that data at different sites in different ways.

The individual who is the network data manager for these systems will have his or her hands full maintaining these types of future systems.

C. INDIVIDUAL DATA NEEDS

All organizations would not necessarily benefit from moving to higher and higher levels of data systems. There are organizations whose use of the data, such as in very high volume transaction processing, may even best be served by file systems. But when different ways of looking at the same data are needed, the data base system is needed. The most frequent implementations today are combinations of class III and class IV. Class V may be a reality in the future, but for now it is more of a concept.

IX. CONCLUSIONS/RECOMMENDATIONS

A. THE PROBLEM

> The maintainer's chief skill, like the surgeon's, is not in making desirable changes but in avoiding undesireable ones. (Any fool can take out an appendix; the trick is to take it out without killing the patient.) [Ref. 9: p. ix]

As has been described the task of computer software maintenance is no easy undertaking and consequently neither is the function of the maintenance manager. A general framework for analyzing this task has been presented to aid in understanding the process.

The central focus of this thesis has been that software evolves. This thesis has examined the internal and external factors involved in the ability of an organization to respond and direct the evolutionary demands on software. In an effort to help the software manager understand software evolution, the authors have concentrated on four topics. Each topic serves as an element of the paradigm of evolution, building upon the last toward the goal of controlling software evolution.

- Historical Perspective: To predict software evolution, the software manager must understand the present and past states of the software system. That understanding is gained through the collection, retention and analysis of data about software evolution.
- The Ability to Predict: Once the historical perspective is achieved, the software manager may predict how various internal and external factors will influence software evolution.

- The Focus of Control: Manpower is the critical resource in software evolution, and thus efforts to control the personnel resource will yield the most substantial influence on software evolution. The key to successful control of the personnel resource is through understanding the nature of the maintenance programmer and how this function is performed.
- The Means of Control: There are several ways to control the influence of personnel on software evolution. The authors chose to focus on the use of software tools, the enforcement of standards and the integration of data as the means of control that would offer the most positive long-range benefits.

B. CONCLUSIONS

1. Historical Data Collection

While data often exists with which a software manager may develop a historical perspective, that data is generally unusable due to a number of deficiencies. Fundamental concepts are not universally defined. The definition of "software maintenance" itself is debated. Concepts such as "program complexity" and "programmer productivity" are defined in largely subjective terms and open to interpretation. Even a physical quantity like "lines of code" cannot be consistently defined.

Without fundamental concepts rigorously defined, metrics to measure the qualities of the software and of the environment cannot be established. The collection, categorization and analysis of data is virtually impossible without a suitable set of software metrics. The characteristic elements of a software system discussed in Chapter IV were presented in a highly subjective manner, and tend to reflect the imprecise nature of contemporary software metrics.

Once a suitable set of software metrics for estimation is derived, data may be collected and analyzed to establish the historical perspective of the software system.

2. Predicting Software Maintenance:

The state of the art of software maintenance cost estimation is hobbled by an incomplete understanding of the factors that influence software evolution. Despite extensive research into software cost estimation, existing development models yield estimates that are, at best, within 20% of the actual cost roughly 80% of the time [Ref. 1: p. 521]. Models designed to estimate software development costs tend to be even more inaccurate when applied to software maintenance [Ref. 28: p. D-16]. Thus, a software manager is forced to employ several techniques and models when attempting to predict future software evolution and estimate the resources required to implement that evolution.

3. Personnel

In a final recognition of the necessity of the maintenance function, managers must value their maintenance personnel. This is a function that will continue to receive more attention as the cost of the maintenance function is shown to be a large percentage of the software life cycle. The goal is to have better and more productive maintenance personnel.

There are three major areas that must not be neglected. These are training, incentives and career progression.

- Maintenance personnel must not be neglected when new techniques, hardware, software, etc. classes are being given. They too must be included so they may be prepared to meet the demands of the future.

- Incentives can come in many forms. Training programs may be an incentive to some career-minded maintainers. Adequate environment (working spaces), the tools and support to do the job, provide a great incentive and can show the maintenance programmer that he or she really does count.

- Finally, the organization must show a valid career progression to which one can aspire. How can the individual reach their career goals within that organization? The military officer has an especially acute problem if he or she wants to consider a career in data processing in the Navy. The officers rotate into and out of the field, creating a very difficult problem of keeping up with the rapidly changing computer technology. The creation of a data processing specialty would alleviate this problem.

4. Tools

In the push to make use of tools in a maintenance environment, the past is most definitely prologue. The standards enforced, the tools used, the structure given in the development of the system will directly effect what can even be attempted in the maintenance phase. When the software to be maintained is an old, assembler language, undocumented system, remedial steps must be taken almost immediately to have the working tools needed for the time when the system bombs. These remedial steps of providing current documentation on these systems can have a two-fold benefit. It becomes a self defense measure to help avoid disaster as well as providing initial training for the maintenance programmers. An example of the very few tools available for this retrofit is presented by Schneider in [Ref. 38].

The progression ideally would assume that the effort in the development group would be toward the use of higher and higher level languages with the comparable larger numbers of tools and environments available. This assumes, as has been shown, that the progression to fourth generation languages with their package of integrated tools will allow a more efficient maintenance function in the future.

The one recommendation the authors make in this area would be for organizations to make better use of user's groups to discuss individual problems and explore the applicability of new software tools. Specific communities containing unique implementations, specialized hardware, unique or obsolete languages, or combinations of these would be aided immeasurablly by contact on a regular basis. This interaction could take the form of phone calls, conferences, newsletters, networking, electronic bulletin boards, etc. Such an interaction would enhance data-processing cohesiveness and offer a ready forum for the exchange of problems and their solutions.

5. Summary

A basic understanding of this software evolution is required for the maintenance manager to be able to anticipate the future; not with crisis management and the dread of impending catastrophe, but with confidence, anticipating where problems may arise and how to meet them. Armed with an accurate software history, the software manager may predict with accuracy future directions for the software system and estimate the resources required to evolve in those directions.

APPENDIX A
TOOLS

A. TOOL CATALOGS AND REFERENCES

Listed below are tool catalogs and references which may be of some use. They contain information on tool availability, functions and features, sources, cost, etc. from [Ref. 51: p. E-1]

1. "DATAPRO Directory of Software," DATAPRO Research Corp., McGraw-Hill.
2. "Software Development Tools", Special Publication 500-88, Raymond C. Houghton, Jr., National Bureau of Standards, March 1982.
3. "Federal Software Exchange Catalog", Federal Software Exchange Center, General Services Administration, Report No. GSA/ADTSC-82/1, January 1982.
4. "FCSC Conversion Tools Survey", Federal Conversion Support Center, General Services Administration, Report No. GSA/FCSC-82/001, October 1982.
5. Software Tool Catalog", Federal Software Testing Center, General Services Administration, Report No. FCTC-82/013, April 1982.
6. AUERBACH Technology Reports, AUERBACH Publishing Inc., 1982.
7. "International Directory of Software, 1980 - 81", CUYB Publications, England, 1980.
8. "The EDP Performance Review -- Ninth Annual Survey of Performance-Related Software Packages", Applied Computer Research, Volume 9, Number 12, December 1981.

9. "Software Engineering Automated Tools Index", Software Research Associates, California, 1981.
10. "Software Tools: Catalogue and Recommendations", TRW Defense and Space Systems Group, 1979.
11. "NBS Software Tools Database", Raymond C. Houghton, Jr. and Karen A. Oakley, NBS-IR-80-2159, National Bureau of Standards, October 1980.
12. "ICP Software Directory - Data Processing Management", P.O. Box 2850, Clinton, Iowa 52732.
13. "AIAA Computer Systems Committee Software Tools Survey", Data & Analysis Center for Software, Rome Air Development Center (RADC), ISISI, Griffiss Air Force Base, NY 13441.
14. "Software Tools Survey", Federal Software Testing Center, General Services Administration, Report No. OSD/FSTC-83/015, June 24, 1983.

B. SOFTWARE MAINTENANCE COST ESTIMATING MODELS

The purpose of this section of the appendix is to briefly summarize selected software maintenance cost estimation models. A rigorous analysis or comparison of the models will not be attempted.

1. Software Lifecycle Model - SLIM

This model is available from Quantitative Software Management, Inc. An automated system, SLIM operates on Hewlett Packard equipment and is in use at the Naval Electronic Systems Command. SLIM is derived from L. Putnam's Life Cycle Model as represented by the Rayleigh distribution. (See Figure 2.3). Courses on the the use of SLIM are offered thrcugh the Department of Defense Computer Institute, Washington, D.C.

2. Constructive Cost Model - COCOMO

The CCCOMO model was developed by Barry W. Boehm and is presented in great detail in his book Software Engineering Economics, [Ref. 1]. COCOMO is easy to use with numerous tables from which the estimator may readily derive the required parametric values. The model algorithm is well-discussed and lends itself well to automation.

3. The Scope of Effcrt Algorithm

This mcdel was developed by G.S. Hoppenstand, II, USN and I. T. Nowak [Ref. 21] at the Naval Security Group Activity, Skaggs Is., California specifically for estimating software maintenance. Their rather unique approach is to analyze the complexity of a given software system, then derive the number of "steps" required to complete an average maintenance task. (This approach is possible largely because the effort of studying the existing system and is the single largest task in performing the software maintenance - Figure 2.1.) Their model then predicts the number of "steps" a military programmer of given experience can complete per year. Thus, the billet requirements may be calculated for a given system.

4. The Model for Estimating Tactical Software Maintenance Requirements

This model was developed by W. H. Merring, III, Capt, USMC as a master's thesis at the Naval Postgraduate School, Monterey, California. The Merring model [Ref. 22] employs "bebugging", a technique of seeding a program with intentional errors to determine the error rate, the detectability of errors, and the maintainability of the program. This technique is used to estimate the corrective maintenance worklcad. Enhancement maintenance is estimated using Halstead's Effort Metric [Ref. 24] as a measure of program complexity. Halstead's metric has been shown to be effective at estimating maintenance costs in unstructured code [Ref. 4: p. 2.14].

LIST OF REFERENCES

1. Boehm, Barry W., Software Engineering Economics, Prentice Hall, Inc., 1981.

2. Lientz, B. P. and Swanson, E. B., Software Maintenance Management, Addison-Wesley, 1980.

3. Pressman, Roger, S., Software Engineering: A Practitioner's Approach, McGraw-Hill Book Co., 1982.

4. Rome Air Development Center Report RADC-TR-80-13, A Review of Software Maintenance Technology, by John D. Donahoo and Dorothy Swearingen, February 1980.

5. Van Horn, Earl C., "Software Must Evolve", Software Engineering, H. Freeman and P. M. Lewis, editors, Academic Press, pp. 209-226, 1980.

6. Rome Air Development Center Report RADC-TR-82-313, Advanced Tools for Software Maintenance, by Jeffery S. Dean and Brian F. McCune, December 1982.

7. Fisher, David A., and Standish, Thomas A., "Initial Thoughts on the Pebbleman Process", Institute for Defense Analysis, January 3, 1979.

8. "Federal Agencies' Maintenance of Computer Programs: Expensive and Undermanaged", General Accounting Office, AFMD-81-25, February 26, 1981.

9. Parikh, Girish and Zvegintzov, Nicholas, "The World of Software Maintenance", Tutorial on Software Maintenance, IEEE Computer Society Press, pp. 1-3, 1983.

10. Fjeldstad, R. K. and Hamlen, W. T., "Application Program Maintenance Study - Report to Our Respondents", by IBM Corporation, DP Marketing Group, White Plains, NY, 1979.

11. McClure, Carma L., Managing Software Development and Maintenance, Van Nostrand Reinhald Company, 1981.

12. Brooks Jr., Fredrick P., The Mythical Man-Month: Essays on Software Engineering, Addison-Wesley, 1975.

13. Putnam, Lawrence H., Tutorial on Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers, IEEE Computer Society Press, section 1, 1980.

14. Richardson, G. I. and Butler, C. W., "Organizational Issues of Effective Maintenance Management", AFIPS Conference Proceedings, National Computer Conference, May 16-19, 1983, AFIPS Press, vol. 52, pp. 155-161, 1983.

15. Lehman, M. M., "Laws and Conservation in Large Program Evolution" in Proceedings, Second Lifecycle Management Workshop, Atlanta, Georgia, pp. 140-145, August 21-22, 1978.

16. Boehm, Barry W., "Economics of Software Maintenance", Proceedings of the Software Maintenance Workshop, at Naval Postgraduate School, Monterey, California, December 6-8, 1983, IEEE Computer Society Press, p. 5, 1983.

17. Air Command and Staff College Report 83-1325, Software Maintenance Cost Estimating, by G. H. Klemas, March 1983.

18. Jones, Carl R. and Ein-Dor, Phillip, Information Resources Management, Elsiever Science, forthcoming.

19. Rome Air Development Center Report RADC-TR-81-144, An Evaluation of Software Cost Estimating Models, by Robert Thibodeau, June 1981.

20. Mohanty, Siba N., "Software Cost Estimation: Present and Future", Software - Practice and Experience, vol, 11, pp. 103-121, 1981.

21. Hoppenstand, G. S. and Nowak, L. J., Management of Naval Security Group Programmer Resources, Naval Security Group Activity, Skaggs Island, California, 1982.

22. Merring III, William H., A Model for Estimating Tactical Software Maintenance Requirements, Master's Thesis, Naval Postgraduate School, Monterey, California, 1982.

23. Glass, Robert L., and Noiseux, Ronald A., Software Maintenance Guidebook, Prentice-Hall, Inc., 1981.

24. Halstead Maurice H., Elements of Software Science, Elsevier North Holland, 1977.

25. Martin, James and McClure, Carma, Software Maintenance: The Problem and Its Solutions, Prentice-Hall, Inc., 1983.

26. Jones, T. C., "Measuring Programming Quality and Productivity", IBM Systems Journal, vol. 17, no. 1, pp. 39-63, 1978.

27. Jones, Caspar, "Estimating Productivity, Quality, and Schedules for Programming Systems", Proceedings of the Software Maintenance Workshop, at Naval Postgraduate School, Monterey, California, December 6-8, 1983, IEEE Computer Society Press, pp. 33-41, 1983.

28. Computer Software Management Subgroup, Proceedings of the Joint Commanders, Joint Policy Coordinating Group on Computer Resource Management, Second Software Workshop, November 1, 1981.

29. Brown, P. J., "Why Does Software Die", Infotech State of the Art Report, series 8, no. 7, pp. 32-45, 1980.

30. Wolverton, R. W., "Software Life Cycle Management-Dynamics Practice", in Proceedings: Second Software Lifecycle Management Workshop, Atlanta, Georgia, pp. 21-29, August 21-22, 1978.

31. "Report of the Department of Defense Joint Service Task Force on Software Problems (U)", AD-A123 449, Department of Defense, Washington, D.C., July 30, 1982.

32. Lyons, Michael J., "Salvaging Your Software Asset (Tools Based Maintenance)", AFIPS Conference Proceedings, National Computer Conference, May 4-7, 1981, AFIPS Press, vol. 50, pp. 337-341, 1981.

33. Booch, Grady, Software Engineering with Ada, The Benjamin/Cummings Publishing Co., Inc., 1983.

34. Schneiderman, Ben, Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishers, Inc., 1980.

35. U. S. Air Force (COMTEC-2000), Computer Technology Forecast and Weapon System Impact Study vol. III, December 1978.

36. Lientz, Bennet P., "Issues in Software Maintenance", University of California, Los Angeles, work partially supported by Office of Naval Research, project number NR 049-345, July 1983.

37. Weinberg, Gerald M., The Psychology of Computer Programming, Van Nostrand Reinhold Company, 1971.

38. Schneider, G. R. Eugenia, Structured Software Maintenance, Master's Thesis, California State University, Chico, 1981.

39. Bronstein, Gary M. and Okamoto, Robert I., "I'm OK, You're OK, Maintenance is OK", Computerworld, January 12, 1981.

40. Canning, Richard G., editor, "That Maintenance 'Iceberg'", EDP Analyzer, Canning Publications, Inc., vol. 10, no. 10, October 1972.

41. Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Transactions on Software Engineering, SE-3(1):60-69, 1977.

42. Ross, D., and Schoman, K., "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, vol. 3, no. 1, pp. 6-15, January 1977.

43. Wasserman, Anthony I., "Information System Design", Journal of the American Society for Information Science, January 1980.

44. Federal Software Testing Center, General Services Administration, Report OSD-82-101, Software Tools Project: A Means of Capturing Technology and Improving Engineering, February, 1982.

45. Notes on Ada Programming Support Environments, Softech, p. 418/4, August 11, 1981.

46. Wolf, M., and others, "The Ada Language System", Computer, p. 38, June 1981.

47. Department of Defense, Requirements for Ada Programming Support Environments: Stoneman, p. 1, February 1980.

48. Gilb, Thomas, "Design by Objectives: Maintainability", Tutorial on Software Maintenance, IEEE Computer Society Press, pp. 167-179, 1983.

49. Department of the Navy, Deputy Chief of Naval Operations (Manpower Personnel and Training), CPNAV P160-S7-84, MAPTIS High Order Language (HOL) Standard, March 1984.

50. Rothnie Jr., J. B. and others, "Introduction to a System for Distributed Databases (SDD-1)", ACM Transactions on Database Systems, vol. 5, no. 1, March 1980.

51. Federal Conversion Support Center, General Services Administration, Report OSD/FCSC-83-006, The Software Improvement Process -- Its Phases and Tasks, Appendices, part 2 of 2, July 1983.

BIBLIOGRAPHY

Cash Jr., James I., McFarlan, F. Warren and McKenney, James L. Corporate Information Systems Management: Text and Cases, Richard D. Irwin, Inc., 1983.

DeMarco, Thomas, Controlling Software Projects, Yordon Press, 1982.

Freeman, Peter and Wasserman, Anthony I., eds., Tutorial on Software Design Techniques, IEEE Computer Society Press, 1983.

Parikh, Girish and Zvegintzov, Nicholas, eds., Tutorial on Software Maintenence, IEEE Computer Society Press, 1983.

Pierce Jr., Charles and Wagner, Rebecca Louise, Software Development Projects: Estimation of Cost and Effort (A Manager's Digest), Master's Thesis, Naval Postgraduate School, 1982.

Tody, Stephen L. and Hodgson, Ray A., Projection of Maximum Software Maintenance Manning Levels, Master's Thesis, Naval Postgraduate School, 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, Califcrnia 93943	2
3.	Cdr Dean Guyer, SC, USN, Code 54Gu Administrative Sciences Department Naval Postgraduate School Monterey, Califcrnia 93943	1
4.	LT Brenda J. Sullivan, USN Naval Military Personnel Command ATTN: (NMPC-47) Navy Department Washington, D.C. 20370	1
5.	LT Brian Hendersen, USN Naval Security Group Activity Fort George G. Meade, MD 20755	1
6.	Professor Carl R. Jones, Code 54Js Administrative Sciences Department Naval Postgraduate School Monterey, Califcrnia 93943	1
7.	Curricular Office - Code 37 Computer Technology Naval Postgraduate School Monterey, Califcrnia 93943	1

1 3 3 3 7 5

210760

Thesis
H43923
c.1

Henderson
Management aspects of
software maintenance.

210760

Thesis
H43923
c.1

Henderson
Management aspects of
software maintenance.

Management aspects of software maintenan
3 2768 001 91828 7
DUDLEY KNOX LIBRARY